ALGORITHMES ET STRUCTURES DE DONNÉES GÉNÉRIQUES

Cours et exercices corrigés en langage C

Michel Divay
Professeur à l'université Rennes 1

2e édition



Illustration de couverture : *Lionel Auvergne*

ISBN 2 10 007450 4

Table des matières

Avant-propos

Ce livre suppose acquis les concepts de base de la programmation tels que les notions de constantes, de types, de variables, de tableaux, de structures, de fichiers et de découpage en fonctions d'un programme. Il présente des notions plus complexes très utilisées en conception de programmes performants sur ordinateur.

Le chapitre 1 introduit la notion de récursivité des procédures et de récursivité des données conduisant à la notion d'allocation dynamique et de pointeurs. Il introduit ensuite la notion de découpage d'une application en modules communiquant grâce à des interfaces basées sur des appels de fonction. Le module devient un nouveau type de données : l'utilisateur du module n'accède pas directement aux données du module qui sont masquées et internes au module. On parle alors d'encapsulation des données qui sont invisibles de l'extérieur du module. Le type ainsi défini est un type abstrait de données (TAD) pour l'utilisateur qui communique uniquement par un jeu de fonctions de l'interface du module. Cette notion est constamment mise en application dans les programmes de ce livre.

Le chapitre 2 présente la notion de listes, très utilisée en informatique dès lors que l'information à gérer est sujette à ajouts ou retraits en cours de traitement. Un module est défini avec des opérations de base sur les listes indépendantes des applications. Des exemples de mise en œuvre sont présentés, ainsi que des variantes des listes (circulaires, symétriques) et des mémorisations en mémoire centrale ou secondaire (fichiers).

Le chapitre 3 définit la notion d'arbres (informations arborescentes). La plupart des algorithmes utilisent la récursivité qui s'impose pleinement pour le traitement

des arbres. Les algorithmes sont concis, naturels, faciles à concevoir et à comprendre dès lors que le concept de récursivité est maîtrisé. De nombreux exemples concrets sont donnés dans ce but. La fin du chapitre présente les arbres ordonnés (les éléments sont placés dans l'arbre suivant un critère d'ordre) pouvant servir d'index pour retrouver rapidement des informations à partir d'une clé. En cas d'ajouts ou de retraits, on peut être amené à réorganiser la structure d'un arbre (le rééquilibrer) pour que les performances ne se dégradent pas trop.

Le chapitre 4 traite de la notion de tables : retrouver une information à partir d'une clé l'identifiant de manière unique. Plusieurs possibilités sont passées en revue en précisant leurs avantages et leurs inconvénients. Plusieurs techniques de hachage (hash-coding) sont analysées sur des exemples simples.

Le chapitre 5 est consacré à la notion de graphes et à leurs mémorisations sous forme de matrices ou de listes d'adjacence. Il donne plusieurs algorithmes permettant de parcourir un graphe ou de trouver le plus court chemin pour aller d'un point à un autre. Là également récursivité et allocation dynamique sont nécessaires.

Les algorithmes présentés sont écrits en C et souvent de manière complète, ce qui permet au lecteur de tester personnellement les programmes et de *jouer avec* pour en comprendre toutes les finesses. Jouer avec le programme signifie être en mesure de le comprendre, de faire des sorties intermédiaires pour vérifier ou expliciter certains points et éventuellement être en mesure de l'améliorer en fonction de l'application envisagée. Les programmes présentés font un minimum de contrôles de validité de façon à bien mettre en évidence l'essentiel des algorithmes. Les algorithmes pourraient facilement être réécrits dans tout autre langage autorisant la modularité, la récursivité et l'allocation dynamique. Le codage est secondaire ; par contre la définition des fonctions de base pour chaque type de structures de données est fondamentale. Chaque structure de données se traduit par la création d'un nouveau type (Liste, Nœud, Table, Graphe) et de son interface sous la forme d'un jeu de fonctions d'initialisation, d'ajout, de retrait, de parcours de la structure ou de fonctions plus spécifiques de la structure de données. Des menus de tests et de visualisation permettent de voir évoluer la structure. Ils donnent de plus des exemples de mise en œuvre des nouveaux types créés.

Les programmes sont **génériques** dans la mesure où chaque structure de données (liste, arbre, table, etc.) peut gérer (sans modification) des objets de types différents (une liste de personnes, une liste de cartes, etc.).

L'ensemble des notions et des programmes présentés constitue une boîte à outils que le concepteur de logiciels peut utiliser ou adapter pour résoudre ses problèmes.

Certains des programmes présentés dans ce livre peuvent être consultés à l'adresse suivante : www.iut-lannion.fr/MD/MDLIVRES/LivreSDD.

Vous pouvez adresser vos remarques à l'adresse électronique suivante :

Michel.Divay@univ-rennes1.fr

D'avance merci.

Michel Divay

Chapitre 1

Récursivité, pointeurs, modules

Ce premier chapitre présente la notion de récursivité, notion très utilisée en programmation, et qui permet l'expression d'algorithmes concis, faciles à écrire et à comprendre. La récursivité peut toujours être remplacée par son équivalent sous forme d'itérations, mais au détriment d'algorithmes plus complexes surtout lorsque les structures de données à traiter sont elles-mêmes de nature récursive. La première partie de ce chapitre présente la récursivité des procédures sur des exemples simples. La seconde partie présente des structures de données récursives et introduit la notion d'allocation dynamique et de pointeurs. La troisième partie présente la notion de découpage d'application en modules.

1.1 RÉCURSIVITÉ DES PROCÉDURES : DÉFINITION

La récursivité est une méthode de description d'algorithmes qui permet à une procédure de s'appeler elle-même (directement ou indirectement). Une notion est récursive si elle se contient elle-même en partie, ou si elle est partiellement définie à partir d'elle-même.

L'expression d'algorithmes sous forme récursive permet des descriptions concises et naturelles. Le principe est d'utiliser, pour décrire l'algorithme sur une donnée **D,** l'algorithme lui-même appliqué à un ou plusieurs sous-ensembles de **D,** jusqu'à ce que le traitement puisse s'effectuer sans nouvelle décomposition. Dans une procédure récursive, il y a deux notions à retenir :

- la procédure s'appelle elle-même : on recommence avec de nouvelles données.
- il y a un test de fin : dans ce cas, il n'y a pas d'appel récursif. Il est souvent préférable d'indiquer le test de fin des appels récursifs en début de procédure.

```
void p (...) {

  if (fin) {
    ...                             pas d'appel récursif (partie "alors")
  } else {
    ...
    p (...);                        la procédure p s'appelle elle-même
    ...                             une ou plusieurs fois (partie "sinon")
  }
}
```

Ainsi, si on dispose des fonctions *void avancer (int lg) ;* qui trace un segment de longueur lg sur l'écran dans la direction de départ, et de la fonction *void tourner (int d) ;* qui change la direction de départ d'un angle de d degrés, on peut facilement tracer un carré sur l'écran en écrivant la fonction récursive suivante :

```
void carre (int lg) {
  avancer (lg);
  tourner (90);
  carre (lg);      // recommencer
}
```

Cependant, la fonction ne comporte pas de test d'arrêt. On recommence toujours la même fonction *carre()*. Le programme boucle. Cet exemple montre la nécessité de la condition d'arrêt des appels récursifs.

1.2 EXEMPLES DE FONCTIONS RÉCURSIVES

1.2.1 Exemple 1 : factorielle

La factorielle d'un nombre n donné est le produit des nombres entiers inférieurs ou égaux à ce nombre n. Cette définition peut se noter de différentes façons.

Une première façon consiste à donner des exemples et à essayer de généraliser.

0! = 1

1! = 1

2! = 1 x 2

3! = 1 x 2 x3

n! = 1 si n = 0

n! = 1 * 2 * ... * (n-1) * n si n > 0

Une deuxième notation plus rigoureuse fait appel à la récurrence.

n! = 1 si n = 0

n! = n * (n-1)! si n > 0

n! se définit en fonction d'elle-même (n-1)!

La fonction *factorielle (n)* permet de calculer la factorielle de n. Cette fonction est récursive et se rappelle une fois en factorielle (n-1). Il y a fin des appels récursifs lorsque n vaut 0.

```
/* factorielle.cpp
   Calcul récursif de la factorielle d'un entier n >= 0 */

#include <stdio.h>  // printf, scanf

// version 1 pour explication
long factorielle (int n) {
  if (n == 0) {
    return 1;
  } else {
    long y = factorielle (n-1);
    return n*y;
  }
}

void main () {
  printf ("Entier dont on veut la factorielle (n<=14) ? ");
  int n; scanf ("%d", &n);

  printf ("Factorielle %d  est : %ld\n", n, factorielle (n) );
}
```

Avec 16 bits, n doit être compris entre 0 et 7 $0! = 1$, $7! = 5\,040$
Avec 32 bits, n doit être compris entre 0 et 14 $14! = 1\,278\,945\,280$

	1	2	3	4

factorielle (3);	n = 3;
	y = factorielle (2);
	
	return 3*2;
		n = 2;
		y = factorielle (1);
		
		return 2*1;
			n = 1;
			y = factorielle (0);
			
			return 1*1;
				n = 0;
				return 1;

Figure 1 Appels récursifs pour factorielle (3).

Pour calculer factorielle(3) sur la Figure 1, il faut calculer factorielle(2). Pour calculer factorielle(2), il faut calculer factorielle(1). Pour calculer factorielle(1), il faut connaître factorielle(0). factorielle(0) vaut 1. On revient en arrière terminer la fonction au niveau 3, puis 2, puis 1.

Il y a autant d'occurrences des variables n et y que de niveaux de récursivité. Il y a création, à chaque niveau, d'un nouvel environnement comprenant les paramètres et les variables locales de la fonction. Cette gestion des variables est invisible à l'utilisateur et effectuée automatiquement par le système si le langage admet la récursivité.

En fait y est ajouté dans la fonction *factorielle()* pour mieux expliquer la récursivité. Les deux instructions long y = factorielle(n-1) ; et return n*y ; peuvent être remplacées de manière équivalente et plus concise par return n*factorielle(n-1).

```
// calcul récursif de factorielle
// 32 bits : limite = 14!
// version finale
long factorielle (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n*factorielle (n-1);
  }
}
```

La procédure itérative donnée ci-dessous est plus performante pour le calcul de factorielle (mais moins naturelle). Il n'y a ni appels en cascade de la fonction, ni environnements multiples des variables. Sur cet exemple, la récursivité ne s'impose pas, et les deux versions (récursive et itérative) ont leurs avantages et leurs inconvénients.

```
/* factorielleIter.cpp
   Calcul itératif de la factorielle d'un entier n >= 0 */

#include <stdio.h>

long factorielleIter (int n) {
  long f = 1;
  for (int i=1; i<=n; i++) f = f * i;
  return f;
}

void main () {
  printf ("Entier dont on veut la factorielle ? ");
  int n; scanf ("%d", &n);
  printf ("Factorielle %d  est : %ld\n", n, factorielleIter (n) );
}
```

1.2.2 Exemple 2 : nombres de Fibonacci

La suite des nombres de Fibonacci se définit comme suit :

$f_o = 0$

$f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$ si $n > 1$

n	0	1	2	3	4	5	6	7	8
f_n	0	1	1	2	3	5	8	13	21

Figure 2 Valeurs de f_n pour n de 0 à 8.

On peut formuler cette suite sous forme de fonction (n >= 0) :
fibonacci (n) = n   si n <= 1
fibonacci (n) = fibonacci (n-1) + fibonacci (n-2)   si n > 1

Exemples de calculs
fibonacci (10) = 55
fibonacci (20) = 6 765
fibonacci (30) = 83 204

Le programme C récursif correspondant est donné ci-dessous.

```
/* fibonacci.cpp
   calcul de fibonacci d'un entier n >= 0 */

#include <stdio.h>

// version 1 pour explication
long fibonacci (int n) {
  if (n <= 1) {
    return n;
  } else {
    long x = fibonacci (n-1);
    long y = fibonacci (n-2);
    return x + y;
  }
}

void main () {
  printf ("Fibonacci de n ? ");
  int n; scanf ("%d", &n);
  printf ("Fibonacci de %d est %ld\n", n, fibonacci (n) );
}
```

Une fonction récursive peut s'appeler elle-même plusieurs fois avec des paramètres différents. La fonction *fibonacci (n)* s'appelle récursivement 2 fois en fibonacci (n-1) et fibonacci (n-2).

	1	2	3
fibonacci (3);	n = 3; x = fibonacci (2);		
		n = 2; x = fibonacci (1);	n = 1; return 1;
		y = fibonacci (0);	n = 0; return 0;
		return 1+0;	
	y = fibonacci (1);	n = 1; return 1;	
	return 1+1;		

Figure 3 Appels récursifs pour fibonacci (3).

Pour calculer fibonacci(3) sur la Figure 3, il faut calculer fibonacci(2) et fibonacci(1). Pour calculer fibonacci(2), il faut calculer fibonacci(1) et fibonacci(0). L'exécution sur la Figure 3 se déroule ligne par ligne, et pour chaque ligne, de la gauche vers la droite.

En fait x et y sont inutiles et ajoutés uniquement pour mieux expliquer la récursivité. La version suivante de *fibonacci()* est équivalente et plus concise.

```
// version définitive
long fibonacci (int n) {
  if (n <= 1) {
    return n;
  } else {
    return fibonacci (n-1) + fibonacci (n-2);
  }
}
```

La fonction récursive est inefficace : les calculs sont répétés un grand nombre de fois. L'appel de la fonction est abrégé en *fib()* sur la Figure 4.

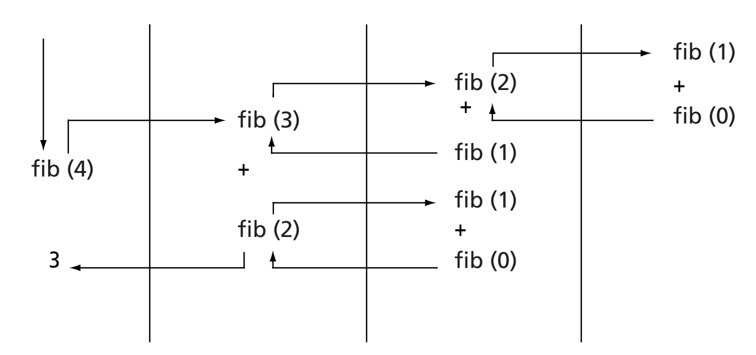


Figure 4 Appels récursifs pour fibonacci(4) (abrégé en fib(4)).

Pour calculer fib(4) sur la Figure 4, il faut calculer fib(3) et fib(2). Pour calculer fib(3), il faut calculer fib(2) et fib(1), etc.

fib(2) est évalué 2 fois,
fib(1) est évalué 3 fois,
fib(0) est évalué 2 fois.

Il est facile de trouver un algorithme **itératif** plus performant, mais moins naturel à écrire.

```
/* fibonacciIter.cpp
   calcul de fibonacci itératif d'un entier n >= 0 */

#include <stdio.h>

// f(0) = 0; f(1) = 1; f(n) = f(n-1) + f (n-2);
long fibonacciIter (int n) {
  long fnm2 = 0;  // fibonacci de n moins 2
  long fnm1 = 1;  // fibonacci de n moins 1
```

```
  long fn;        // fibonacci de n

  if (n <= 1) {
    fn = n;
  } else {
    for (int i=2; i<=n; i++) {
      fn   = fnm1 + fnm2;
      fnm2 = fnm1;
      fnm1 = fn;
    }
  }
  return fn;
}

void main () {
  printf ("Fibonacci de n ? ");
  int n; scanf ("%d", &n);

  printf ("Fibonacci de %d est %ld\n", n, fibonacciIter (n) );
}
```

1.2.3 Exemple 3 : boucles récursives

Une boucle peut s'écrire sous forme récursive. Il faut réaliser une action (écriture par exemple), et recommencer avec une nouvelle valeur de l'indice de boucle. La boucle peut être croissante ou décroissante suivant que l'action est réalisée avant ou après l'appel récursif.

```
/* boucles.cpp  boucles sous forme récursive */

#include <stdio.h>

// boucle décroissante de n à 1
void boucleDecroissante (int n) {
  if (n > 0) {
    printf ("BoucleDecroissante valeur de n : %d\n", n);
    boucleDecroissante (n-1);
  }
}

// boucle croissante de 1 à n
void boucleCroissante (int n) {
  if (n > 0) {
    boucleCroissante (n-1);
    printf ("BoucleCroissante valeur de n : %d\n", n);
  }
}

void main() {
  boucleDecroissante (5);
  printf("\n");
  boucleCroissante (5);
  printf("\n");
}
```

Résultats de l'exécution :

```
boucleDecroissante valeur de n : 5
boucleDecroissante valeur de n : 4
boucleDecroissante valeur de n : 3
boucleDecroissante valeur de n : 2
boucleDecroissante valeur de n : 1

boucleCroissante valeur de n : 1
boucleCroissante valeur de n : 2
boucleCroissante valeur de n : 3
boucleCroissante valeur de n : 4
boucleCroissante valeur de n : 5
```

Exercice 1 - Boucle sous forme récursive

Écrire une fonction récursive *void boucleCroissante (int d, int f, int i) ;* qui effectue une boucle croissante de l'indice d de départ jusqu'à l'indice f de fin par pas de progression de i (i : entier positif). Exemple : boucleCroissante (5, 14, 2); effectue la boucle avec l'indice de départ 5, en progressant de 2 à chaque fois, jusqu'à l'indice 14, soit : 5, 7, 9, 11 et 13.

1.2.4 Exemple 4 : numération

La fonction récursive *long convertirEnBase10 (long n, int base);* convertit un nombre n>=0 écrit en base *base* (avec des chiffres compris entre 0 et 9) en un nombre en base 10. Ainsi convertirEnBase10 (100, 2) fournit 4 ; convertirEnBase10 (137, 11) fournit 161 ; convertirEnBase10 (100, 16) fournit 256.

```
/* convertir.cpp convertir de façon récursive
                 un nombre d'une base dans une autre base */

#include <stdio.h>

// la fonction convertit en base 10, n en base "base";
// n est composé des chiffres de 0 à 9; n >= 0, base > 0
long convertirEnBase10 (long n, int base) {
  long quotient = n / 10;
  long reste    = n % 10;
  if (quotient == 0) {
    return reste;
  } else {
    return convertirEnBase10 (quotient, base) * base + reste;
  }
}
```

La Figure 5 indique le déroulement de l'exécution de la fonction *convertirEnBase10 ()* pour la conversion de 137 en base 11. L'exécution se déroule, pour chaque ligne du

tableau, de haut en bas et de gauche à droite. L'appel de la fonction est abrégé en *convert()* sur le schéma.

	1	2	3

convert (137, 11);	n = 137;
	base = 11;
	quotient = 13;
	reste = 7;
	convert (13, 11);
		n = 13;
		base = 11;
		quotient = 1;
		reste = 3;
		convert (1, 11);
			n = 1;
			base = 11;
			quotient = 0;
			reste = 1;
			return 1;

		1*11+3; return 14;	
	14*11+7; return 161;		

Figure 5 Appels récursifs pour la conversion de 137 en base 11.

La fonction récursive *void convertirEnBaseB (long n, int base) ;* convertit un entier n en base 10, en un nombre en base *base*.

```
// convertir en base "base", un nombre n en base 10

// (n>=0; 1<base<=16)
void convertirEnBaseB (long n, int base) {
  static char chiffre[] = "0123456789ABCDEF";
  long quotient = n / base;
  long reste    = n % base;
  if (quotient != 0) convertirEnBaseB (quotient, base);
  printf ("%c", chiffre [reste]);
}

void main () {
  long n    = 100;
  int  base = 2;
  printf ("%ld en base %2d = %ld en base 10\n",
           n, base, convertirEnBase10 (n, base) );

  n    = 137;
  base = 11;
  printf ("%ld en base %2d = %ld en base 10\n",
           n, base, convertirEnBase10 (n, base) );

  n    = 100;
  base = 16;
  printf ("%ld en base %2d = %ld en base 10\n",
           n, base, convertirEnBase10 (n, base) );

  printf ("\n0   en base 2 : "); convertirEnBaseB (0,   2);
  printf ("\n25  en base 2 : "); convertirEnBaseB (25,  2);
  printf ("\n255 en base 2 : "); convertirEnBaseB (255, 2);
  printf ("\n256 en base 2 : "); convertirEnBaseB (256, 2);
  printf ("\n10  en base 8 : "); convertirEnBaseB (10,  8);
  printf ("\n161 en base 11: "); convertirEnBaseB (161, 11);
  printf ("\n26  en base 16: "); convertirEnBaseB (26,  16);
  printf ("\n255 en base 16: "); convertirEnBaseB (255, 16);
  printf ("\n256 en base 16: "); convertirEnBaseB (256, 16);
  printf ("\n");
}
```

Résultats du programme précédent :

```
100 en base  2 = 4 en base 10
137 en base 11 = 161 en base 10
100 en base 16 = 256 en base 10

0   en base 2 : 0
25  en base 2 : 11001
255 en base 2 : 11111111
256 en base 2 : 100000000
10  en base 8 : 12
161 en base 11: 137
26  en base 16: 1A
255 en base 16: FF
256 en base 16: 100
```

1.2.5 Exemple 5 : puissance nième d'un nombre

Pour calculer un nombre x^n (x à une puissance entière n, n>=0), on peut effectuer n fois la multiplication de x par lui-même, sauf si n vaut zéro auquel cas x^0 vaut 1. Ainsi, pour calculer 3^4 (x=3 à la puissance n=4), on multiplie 3*3*3*3. Une autre façon plus efficace de faire est de calculer 3^2 (3*3), et de multiplier ce résultat par lui-même 3*3. Ainsi, si n est pair, le calcul de x^n se ramène au calcul de $x^{n/2}$ que l'on multiplie par lui-même. Si n est impair, prenons le cas de 3^5 (x=3 à la puissance n=5), on calcule de même 3^2 que l'on multiplie par lui-même 3^2 puis par 3. On applique récursivement la même méthode pour calculer 3^2. Le programme qui en découle est donné ci-dessous. On pourrait ajouter un cas particulier pour n=1, mais il n'est pas nécessaire du point de vue algorithmique.

```
/* puissance.cpp  puissance nième entière d'un nombre réel */

#include <stdio.h>

// puissance nième d'un nombre réel x (n entier >= 0)
double puissance (double x, int n) {
  double r;  // résultat

  if (n == 0) {
    r = 1.0;
  } else {
    r = puissance (x, n/2);
    if (n%2 == 0) {
      r = r*r;      // n pair
    } else {
      r = r*r*x;    // n impair
    }
  }
  return r;
}
```

```
void main () {
  printf ("3   puissance 4  %.2f\n", puissance (3, 4) );
  printf ("3.5 puissance 5  %.2f\n", puissance (3.5, 5) );
  printf ("2   puissance 0  %.2f\n", puissance (2, 0) );
  printf ("2   puissance 1  %.2f\n", puissance (2, 1) );
  printf ("2   puissance 2  %.2f\n", puissance (2, 2) );
  printf ("2   puissance 3  %.2f\n", puissance (2, 3) );
  printf ("2   puissance 10 %.2f\n", puissance (2, 10) );
  printf ("2   puissance 32 %.2f\n", puissance (2, 32) );
  printf ("2   puissance 64 %.2f\n", puissance (2, 64) );
}
```

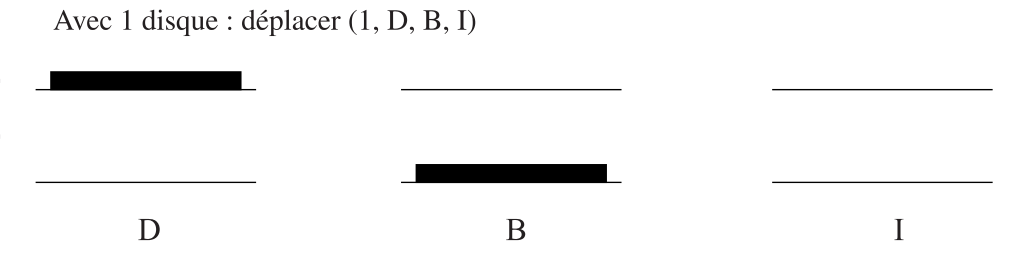
Exemples de résultats :

```
3   puissance 4  81.00
3.5 puissance 5  525.22
2   puissance 0  1.00
2   puissance 1  2.00
2   puissance 2  4.00
2   puissance 3  8.00
2   puissance 10 1024.00
2   puissance 32 4294967296.00
2   puissance 64 18446744073709551616.00
```

1.2.6 Exemple 6 : Tours de Hanoi

Les « Tours de Hanoi » est un jeu où il s'agit de déplacer un par un des disques superposés de diamètre décroissant d'un socle de départ D sur un socle de but B, en utilisant éventuellement un socle intermédiaire I. Un disque ne peut se trouver au-dessus d'un disque plus petit que lui.

Le schéma de la Figure 6 montre le raisonnement mettant en évidence le caractère récursif de l'algorithme à suivre. Pour déplacer un disque de D vers B, le socle I (intermédiaire) est inutile. Pour déplacer 2 disques, il faut transférer celui qui est au sommet sur le socle I, déplacer le disque reposant sur le socle D vers B, et ramener le disque du socle I au sommet de B. Pour déplacer 3 disques, il faut déplacer les 2 disques en sommet de D vers I (en utilisant B comme intermédiaire), ensuite déplacer le disque reposant sur le socle de D vers B, et ramener les 2 disques mis de côté sur I, en sommet de B. Pour déplacer 3 disques, on utilise 2 fois la méthode permettant de déplacer 2 disques sans entrer dans le détail de ces mouvements, ce qui met en évidence la récursivité.



Avec 2 disques : déplacer (2, D, B, I)

D B I

Avec 3 disques : déplacer (3, D, B, I)

Déplacer (2, D, I, B)

D vers B

Déplacer (2, I, B, D)

D B I

Figure 6 Tours de Hanoi : principe.

D'une manière générale comme l'indique la Figure 7, pour déplacer N disques de D vers B, il faut déplacer les N-1 disques de D vers I en utilisant B comme intermédiaire, déplacer le disque restant de D vers B, ramener les N-1 disques de I vers B en utilisant D comme intermédiaire.

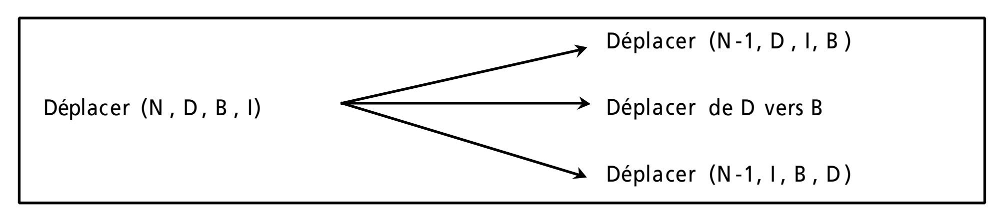


Figure 7 Tours de Hanoi : algorithme.

L'algorithme récursif découle directement du raisonnement suivi ci-dessus.

```
/* hanoi.cpp  tours de Hanoi
              exemple de programme avec deux appels récursifs */

#include <stdio.h>
#include <time.h>   // time()

// déplacer n disques du socle depart vers le socle but
// en utilisant le socle intermédiaire.
void deplacer (int n, char depart, char but, char intermediaire) {
  if (n > 0) {
    deplacer (n-1, depart, intermediaire, but);
    printf  ("Déplacer un disque de %c --> %c\n", depart, but);
    deplacer (n-1, intermediaire, but, depart);
  }
}

void main () {
  printf ("Nombre de disques à déplacer ? ");
  int n; scanf  ("%d", &n);

  //time_t topDebut; time (&topDebut);
  deplacer (n, 'A', 'B', 'C');
  //time_t topFin; time (&topFin);
  //printf ("de %d à %d = %d\n", topDebut, topFin, topFin - topDebut);
}
```

Si n = 3, le résultat de l'exécution de *deplacer()* est le suivant :

```
Déplacer un disque de A--> B
Déplacer un disque de A--> C
Déplacer un disque de B--> C
Déplacer un disque de A--> B
Déplacer un disque de C--> A
Déplacer un disque de C--> B
Déplacer un disque de A--> B
```

Les déplacements sont schématisés sur la Figure 8. Au départ, on a 3 disques de taille décroissante 3, 2, 1 sur le socle A. Il faut les transférer sur le socle B, ce qui est accompli à la dernière ligne du tableau.

	Socle A			Socle B			Socle C		
Départ	3	2	1						
A → B	3	2		1					
A → C	3			1			2		
B → C	3						2	1	
A → B				3			2	1	
C → A	1			3			2		
C → B	1			3	2				
A → B				3	2	1			

Figure 8 Tours de Hanoi : résultats.

Le test de la procédure récursive aurait pu être écrit de manière légèrement différente mais équivalente, correspondant au raisonnement suivant : s'il n'y a qu'un disque, le déplacer, sinon déplacer les N-1 du sommet de D vers I, déplacer le disque restant de D vers B, ramener les N-1 de I vers B.

```
// déplacer n disques du socle depart vers le socle but
// en utilisant le socle intermédiaire.
void deplacer (int n, char depart, char but, char intermediaire) {
  if (n == 1) {
    printf  ("Déplacer un disque de %c --> %c\n", depart, but);
  } else {
    deplacer (n-1, depart, intermediaire, but);
    printf  ("Déplacer un disque de %c --> %c\n", depart, but);
    deplacer (n-1, intermediaire, but, depart);
  }
}
```

On peut évaluer le nombre de déplacements à effectuer pour transférer n disques de D vers B (départ vers but).

si n = 1,

déplacer le disque de D vers B	$c_1 = 1$

sinon

déplacer n-1 disques de D vers I	c_{n-1}
déplacer le disque n de D vers B	1
déplacer n-1 disques de I vers B	c_{n-1}
	$c_n = 2 * c_{n-1} + 1$

Cette équation peut se développer en :
$2^{n-1} + 2^{n-2} + ... 2 + 1$
qui est la somme $2^n - 1$ des n termes d'une progression géométrique de raison 2.

si n = 1	alors	$C_1 = 2^1 - 1$	= 1
si n = 2	alors	$C_2 = 2^2 - 1$	= 3
si n = 3	alors	$C_3 = 2^3 - 1$	= 7 (voir Figure 8)
si n = 4	alors	$C_4 = 2^4 - 1$	= 15
si n = 64	alors	$C_{64} = 2^{64} - 1$	

Les tests suivants utilisant la fonction *time()* fournissant le temps écoulé depuis le 1/1/1970 en secondes permettent de calculer le temps d'exécution (la fonction *printf* de *deplacer()* étant mise en commentaire) :

```
//ordinateur : Pentium III 800 Mhz (Linux Red Hat)
//n                   25   26   27   28   29   30   31   32
//durée en secondes   2    5    8   17   35   67  140  270
```

Avec 64 disques, si un déplacement s'effectue en 60 nanosecondes (valeur approximative calculée ci-dessus), il faut plus de 30 mille ans pour effectuer tous les déplacements.

Exercice 2 - Hanoi : calcul du nombre de secondes ou d'années pour déplacer n disques

Un déplacement se faisant en 60 nanosecondes, faire un programme qui calcule :

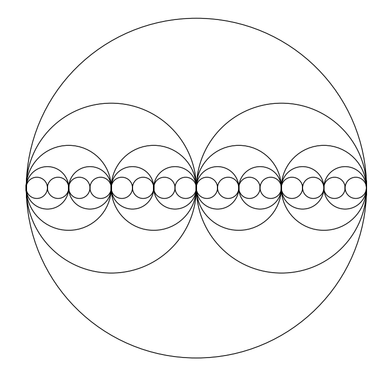
• le nombre de secondes nécessaires pour déplacer de 25 à 32 disques.

• le nombre d'années nécessaires pour déplacer 64 disques.

Utiliser la fonction *time()* fournissant le temps écoulé en secondes depuis le 1/1/1970 pour chronométrer, sur votre ordinateur, le temps d'exécution pour des valeurs de n entre 25 et 32.

1.2.7 Exemple 7 : tracés récursifs de cercles

Le premier dessin de la Figure 9 représente un cercle qui contient deux cercles qui contiennent deux cercles, ainsi de suite, jusqu'à ce que le dessin devenant trop petit, on décide d'arrêter de dessiner. Le dessin peut être qualifié de récursif. Le dessin se contient lui-même, et il y a un arrêt à ce dessin récursif. Ceci est également vrai pour le deuxième dessin contenant trois cercles.



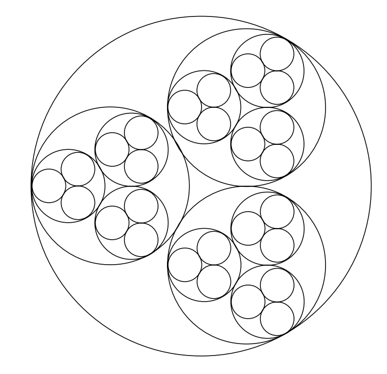


Figure 9 Cercles récursifs.

Le codage des algorithmes de tracé de cercles est très dépendant du système d'exploitation et du compilateur utilisés. Néanmoins, l'algorithme en lui-même est général et peut être codé avec le jeu de fonctions de dessin graphique disponibles. On suppose définie une fonction *void cercle (x, y, r) ;* qui trace un cercle de rayon r centré en (x, y).

Pour la fonction *deuxCercles()*, si le rayon r est supérieur à 10 pixels, on trace un cercle de rayon r en (x, y), et on recommence une tentative de tracé de 2 cercles de rayon pr=r/2 en (x+pr, y), et en (x-pr, y). Si r est inférieur ou égal à 10, on ne fait rien, ce qui arrête les tracés récursifs en cascade.

```
void deuxCercles (int x, int y, int r) {
  if (r > 10) {
    cercle (x, y, r);
    int pr = r / 2;     // petit rayon
    deuxCercles (x+pr, y, pr);
    deuxCercles (x-pr, y, pr);
  }
}
```

> ***Remarque*** : la circonférence du cercle englobant est 2πr, de même que la somme des circonférences des 2 cercles de rayon r/2, ou celle des 4 cercles de rayon r/4.

Pour la fonction *troisCercles()*, le calcul du rayon des cercles inclus et des coordonnées des centres demandent une petite étude géométrique. Soient r, le rayon du cercle englobant, pr, le rayon des 3 cercles inclus et h, la distance entre le centre du grand cercle et le centre d'un des 3 cercles inclus. Les centres des 3 cercles inclus déterminent un triangle équilatéral. Les hauteurs de ce triangle équilatéral déterminent 6 triangles rectangles d'hypoténuse h et de grand côté pr.

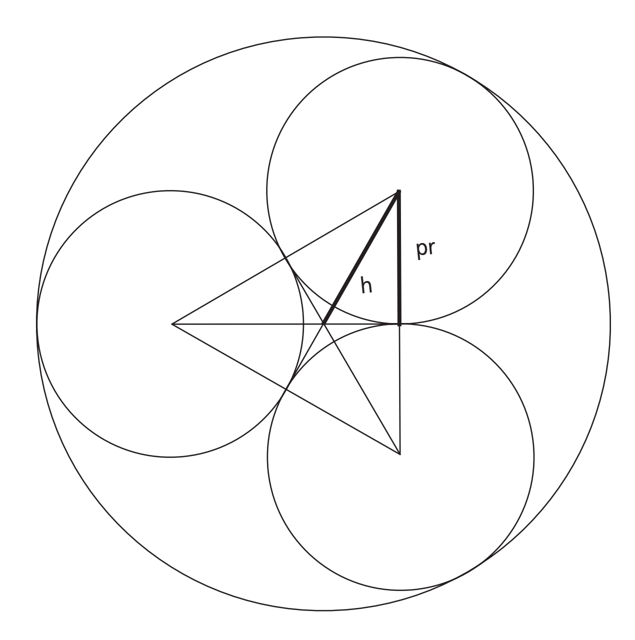


Figure 10 Calcul du rayon des cercles intérieurs.

On a les relations suivantes :

r	= h + pr ;	
pr	= h * sqrt (3)/2 ;	

d'où on peut déduire :

pr	= (2 * sqrt (3) – 3) * r	= 0.4641 * r ;
h	= (4 – 2 * sqrt (3)) * r	= 0.5359 * r ;

Les coordonnées des centres des trois cercles intérieurs s'en déduisent alors facilement.

```
void troisCercles (int x, int y, int r) {
  if (r > 10) {
    cercle (x, y, r);
    //int pr = int ((2*sqrt(3.0)-3)*r);
    int pr = (int) (0.4641*r);
    int h  = r - pr;
    troisCercles (x-h,   y,    pr);
    troisCercles (x+h/2, y+pr, pr);
    troisCercles (x+h/2, y-pr, pr);
  }
}
```

1.2.8 Exemple 8 : tracé d'un arbre

Le dessin de l'arbre de la Figure 11 est obtenu par exécution d'un programme récursif de tracé de segments.

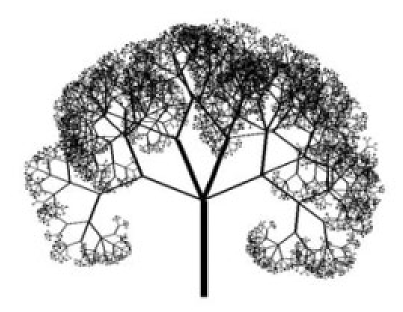


Figure 11 Arbre récursif de tracé de segments.

La fonction *void avance (int lg, int x, int y, int angle, int* nx, int* ny) ;* trace un trait de longueur lg en partant du point de coordonnées (x, y), et avec un angle en degrés angle (voir Figure 12). Le point d'arrivée est un paramètre de sortie (nx, ny). La largeur du trait lg/10 est obtenue en traçant plusieurs traits les uns à côté des autres. La fonction *void traceTrait (x1, y1, x2, y2) ;* trace un trait entre les 2 points (x1, y1) et (x2, y2).

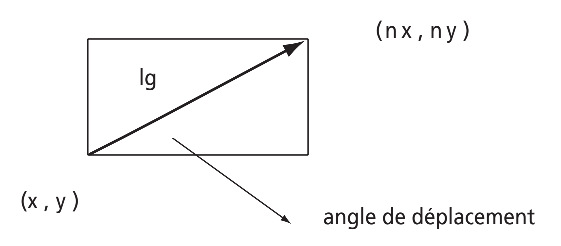


Figure 12 La fonction *avance()*.

```
void avance (int lg, int x, int y, int angle, int* nx, int* ny ) {
  #define PI 3.1416

  *nx = x + (int) (lg * cos (angle*2*PI / 360.));
  *ny = y - (int) (lg * sin (angle*2*PI / 360.));
  traceTrait (x, y, *nx, *ny);

  // l'épaisseur du trait du segment : nb segments juxtaposés
  int nb = lg/10;
  // si nb = 0 ou 1, la boucle n'est pas effectuée
  for (int i=-nb/2; i<nb/2; i++) {
    traceTrait (x+i, y, *nx+i, *ny);
  }
}
```

La fonction *void dessinArbre (int lg, int x, int y, int angle) ;* ajoute aléatoirement à lg une valeur comprise entre 0 et 10 % de lg de façon à introduire une légère dissymétrie dans l'arbre. On trace un trait de longueur lg en partant du point (x, y) dans la direction angle. Si 2*lg/3 est supérieur à un pixel, on génère un nombre aléatoire n valant 2, 3 ou 4, et on appelle **récursivement** la fonction arbre() 2, 3 ou 4 fois avec une longueur à tracer de 2/3 de la longueur de départ et dans des directions réparties entre –π/2 et π/2 de l'angle initial. Sur la Figure 13, on trace un segment de longueur lg, puis 3 segments de longueur 2*lg/3.

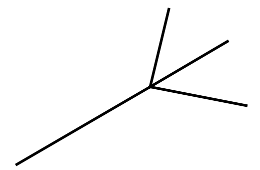


Figure 13 La fonction *dessinArbre()* pour n = 3.

```
// aléatoire entre 0 et max; voir man rand
int aleat (int max) {
  return (int) (rand() % max);
}

void dessinArbre (int lg, int x, int y, int angle) {
  int nx, ny;
  lg = (int) (lg + 0.1 * aleat(lg));
  avance (lg, x, y, angle, &nx, &ny);
  lg = 2*lg / 3;
  if (lg > 1) {
    int n = 2 + aleat (3);
    int d = 180 / n;
    for (int i=1; i<=n; i++) {
      dessinArbre (lg, nx, ny, angle-90 - d/2 + i*d);
    }
  }
}
```

Exercice 3 - Dessin d'un arbre

Modifier la fonction *dessinArbre()* de façon à dessiner le segment terminal en vert et à insérer de façon aléatoire, un fruit rouge (segment ou cercle rouge).

1.2.9 Conclusions sur la récursivité des procédures

On a vu qu'une boucle peut s'écrire sous forme récursive. Quand il n'y a qu'un seul appel récursif, on peut passer à une procédure itérative avec boucle ; c'est le cas des exemples pour factorielle, les nombres de Fibonacci ou les conversions (numération). Par contre, si la procédure divise le problème récursivement en plusieurs sous-problèmes, la récursivité s'impose. Refuser la récursivité, c'est s'obliger à gérer une pile pour retrouver le contexte. Dans le cas de deux appels récursifs par exemple, il y a deux tâches à accomplir. Il faut effectuer le premier appel récursif, en sauvegardant le contexte que l'on retrouve quand on a fini le premier appel récursif et les appels qu'il a engendrés en cascade. La procédure itérative serait beaucoup plus longue et beaucoup moins naturelle. Les procédures des Tours de Hanoi, des dessins des cercles ou de l'arbre s'écrivent, sous forme récursive, de manière concise et naturelle.

1.3 RÉCURSIVITÉ DES OBJETS

Un objet récursif est défini par rapport à lui-même. La construction de procédures récursives est particulièrement appropriée quand la structure des objets manipulés est elle-même récursive.

1.3.1 Rappel sur les structures

Le programme suivant décrit une structure de type Personne et deux variables jules et jacques de ce nouveau type.

```
/* structure1.cpp  rappel sur les structures */

#include <stdio.h>
#include <string.h>

typedef char ch15 [16]; // 15 car. + 0 de fin de chaîne

// le type structure Personne
typedef struct {
  ch15 nom;
  ch15 prenom;
} Personne;

void main () {
  Personne jules;     // jules   variable de type Personne
  Personne jacques;   // jacques variable de type Personne
```

```
    // copier dans la structure jules
    strcpy (jules.nom,       "Durand");
    strcpy (jules.prenom,    "Jules");

    // copier dans la structure jacques
    strcpy (jacques.nom,     "Dupond");
    strcpy (jacques.prenom,  "Jacques");

    printf ("%s %s\n", jacques.nom, jacques.prenom);

  }  // main
```

1.3.2 Exemple de déclaration incorrecte

Le programme suivant est incorrect car il essaie de décrire une personne comme ayant les caractéristiques suivantes : un nom et un prénom, une personne père et une personne mère. La structure est récursive (autocontenue) puisqu'on décrit une personne comme contenant deux variables de type Personne qui contiennent chacune deux variables de type Personne, etc. Cette déclaration est incorrecte car le compilateur ne peut déterminer *a priori* la taille en octets de la structure Personne.

```
/* structure2.cpp
   déclaration incorrecte de structure autocontenue */

#include <stdio.h>
#include <string.h>

typedef char ch15 [16];

// le type Personne se référençant lui-même
// --> erreur de compilation
typedef struct {
  ch15     nom;
  ch15     prenom;
  //Personne pere;    //  --> syntax erreur
  //Personne mere;    //  --> syntax erreur
} Personne;

void main () {
  Personne jules;

  strcpy (jules.nom,    "Durand");
  strcpy (jules.prenom, "Jules");

  printf ("%s %s\n", jules.nom, jules.prenom);
}
```

1.3.3 Structures et pointeurs

Au lieu de décrire un champ de type Personne dans un objet de type Personne, on décrit un **pointeur** sur un objet de type Personne. Un pointeur fournit l'adresse de la personne pointée. Le programme suivant permet la construction de structures de données complexes correspondant à un arbre généalogique. pere est un pointeur de Personne sur la personne père. Ainsi, pour chaque Personne, on a l'adresse en mémoire de son père et de sa mère. L'absence de pointeur (père ou mère inconnus sur l'exemple) se note NULL.

Pour bien mettre en évidence les différences d'allocation et de notation, deux structures sont allouées dynamiquement (en cours d'exécution du programme à l'aide de *malloc()*) : celles pointées par jules et jacques, alors que la structure berthe est allouée statiquement en début de programme (voir Figure 14).

jules->pere se lit : le (champ) pere de la structure pointée par jules.
berthe.pere se lit : le (champ) pere de la structure berthe.

jacques->pere = jules ; Mettre dans le (champ) pere de la structure pointée par jacques, le pointeur jules.

jacques->mere = &berthe ; Mettre dans le champ mere de la structure pointée par jacques, l'adresse de la structure berthe.

p = jules ; Le pointeur p repère la même Personne que le pointeur jules.

**p = *jacques ;* On range à l'adresse pointée par p, les informations (champs) de la Personne se trouvant à l'adresse pointée par jacques.

```
/* structure3.cpp  pointeurs de personnes */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef char ch15 [16];

typedef struct personne {
  ch15      nom;
  ch15      prenom;
  struct personne* pere;   // pere est un pointeur de Personne
  struct personne* mere;   // mere est un pointeur de Personne
} Personne;

void main () {
  Personne* jules;      // pour allocation dynamique
  Personne* jacques;    // pour allocation dynamique

  Personne  berthe;     // allocation statique

  // Réserver de la place en mémoire pour une personne,
  // et mettre l'adresse de la zone allouée
  // dans le pointeur de Personne jules
  //jules   = (Personne*) malloc (sizeof (Personne));  // en C
  //jacques = (Personne*) malloc (sizeof (Personne));
  jules   = new Personne();  // en C++
  jacques = new Personne();

  strcpy (jules->nom,       "Durand");
  strcpy (jules->prenom,    "Jules");
  jules->pere   = NULL;
  jules->mere   = NULL;
```

```
    strcpy (jacques->nom,     "Durand");
    strcpy (jacques->prenom,  "Jacques");
    jacques->pere = jules;
    jacques->mere = &berthe;

    strcpy (berthe.nom,       "Dupond");
    strcpy (berthe.prenom,    "Berthe");
    berthe.pere   = NULL;
    berthe.mere   = NULL;

    printf ("Nom de berthe   : %15s\n", berthe.nom);
    printf ("Nom de jacques  : %15s\n", jacques->nom);
    printf ("\n");

    printf ("Père de Jacques : %15s %15s\n",
      jacques->pere->nom, jacques->pere->prenom);
    printf ("Mère de Jacques : %15s %15s\n",
      jacques->mere->nom, jacques->mere->prenom);
    printf ("\n");

    Personne* p;
    p = jules; // p pointe sur le même objet que jules
    printf ("Personne p : %15s %15s\n", p->nom, p->prenom);

    p  = (Personne*) malloc (sizeof (Personne));
    *p = *jacques; // L'objet pointé par p reçoit le contenu
                   // de l'objet pointé par jacques
    printf ("Personne p : %15s %15s\n", p->nom, p->prenom);

    free (jules);
    free (jacques);
    free (p);
  }
```

Les pointeurs permettent de relier entre elles les différentes structures correspondant aux différentes personnes. Le père de jacques, c'est jules ; la mère de jacques, c'est berthe (voir Figure 14). On peut facilement parcourir les différentes structures de données grâce aux pointeurs et retrouver par exemple, à partir du pointeur jacques, le nom de la mère de jacques.

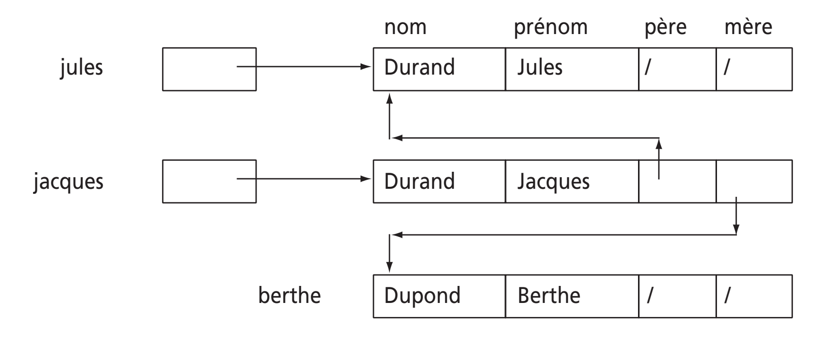


Figure 14 La structure de données des relations entre personnes.

1.3.4 Opérations sur les pointeurs

Soient p et q, deux pointeurs sur un objet de type t.

a) **création** dynamique d'un objet

`t* p = (t*) malloc (sizeof (t) );`   // en C

crée un objet de type t ; p pointe sur cet objet. p contient l'adresse de l'objet créé ; p est un pointeur de t.

Le langage C++ permet de créer un objet (une structure) de type t à l'aide de *new()* comme suit : `t* p = new t();`

b) **affectation** de pointeurs

p = NULL ;   NULL indique un pointeur nul (absence de pointeur)

p = q ;   p et q repèrent le même objet

c) **affectation** de zones pointées

*p = *q ;   mettre à l'adresse pointée par p,
ce qu'il y a à l'adresse pointée par q.

d) **comparaison** de pointeurs (== et !=)

if (p == q) {...   si p et q repèrent le même objet ...

if (p != q) {...   si p et q ne repèrent pas le même objet ...

On peut tester si un pointeur est supérieur ou inférieur à un autre pointeur notamment lorsqu'on parcourt un tableau d'éléments de type t. Le pointeur courant doit être compris entre l'adresse de début et l'adresse de fin du tableau. Pour deux objets indépendants alloués par *malloc()*, seules égalité et inégalité ont un sens.

e) **accès** à l'élément pointé

p→nom = "Durand";   le champ nom de la structure pointée par p

p→svt→nom = "Dupond";

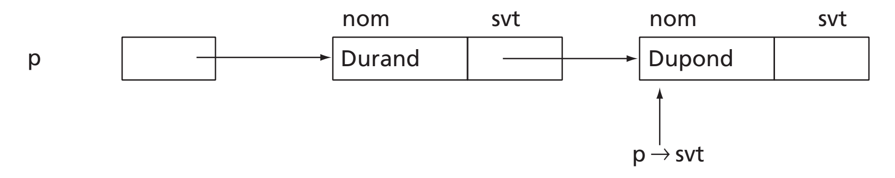


Figure 15 Le champ nom de la structure pointée par p→svt.

f) **suppression de la zone allouée dynamiquement**

free (p) ; rend au système d'exploitation l'espace mémoire occupé par l'objet pointé par p, et alloué par *malloc(),* ou *new()*.

g) **addition, soustraction de pointeurs**

Si p pointe un objet de type t, p+1 pointe l'objet suivant de type t, de même que p++.

Si p et q sont 2 pointeurs de type t, p-q indique le nombre d'objets de type t entre p et q.

Sur l'exemple précédent concernant le type Personne, on pourrait rajouter les instructions suivantes pour tester ces cas d'additions de constantes à un pointeur ou de soustraction de deux pointeurs. On déclare un tableau de Personne que l'on initialise partiellement. ptc est un pointeur courant de Personne, donc de type Personne*, initialisé sur le début du tableau.

```
Personne  tabPers [5];
Personne* ptc = tabPers;

strcpy (tabPers [0].nom, "P0");
strcpy (tabPers [1].nom, "P1");
strcpy (tabPers [2].nom, "P2");
strcpy (tabPers [3].nom, "P3");
strcpy (tabPers [4].nom, "P4");

printf ("%s\n", ptc->nom);   // P0
ptc++;
printf ("%s\n", ptc->nom);   // P1
ptc += 2;
printf ("%s\n", ptc->nom);   // P3

printf ("%d\n", &tabPers[4] - &tabPers[0]);   //  4
printf ("%d\n", &tabPers[0] - &tabPers[4]);   // -4
```

1.4 MODULES

1.4.1 Notion de module et de type abstrait de données (TAD)

D'une manière générale, un module est une unité constitutive d'un ensemble. En algorithmique, un module est un ensemble de fonctions traitant des données communes. Les objets (constantes, variables, types, fonctions) déclarés dans la partie interface sont accessibles de l'extérieur du module, et sont utilisables dans un autre programme (un autre module ou un programme principal). Il suffit de référencer le module pour avoir accès aux objets de sa partie interface. Celle-ci doit être la plus réduite possible, tout en donnant au futur utilisateur un large éventail de possibilités d'utilisation du module. Les déclarations de variables doivent être évitées au maximum. On peut toujours définir une variable locale au module à laquelle on accède ou que l'on modifie par des appels de fonctions de l'interface. On parle alors *d'encapsulation* des données qui sont invisibles pour l'utilisateur du module et seulement accessibles à travers un jeu de fonctions. L'utilisateur du module n'a pas besoin de savoir comment sont mémorisées les données ; le module est pour lui un *type abstrait de données (TAD)*. Du reste, cette mémorisation locale peut évoluer, elle n'affectera pas les programmes des utilisateurs du module dès lors que les prototypes des fonctions d'interface restent inchangés.

	module
Partie interface visible donc accessible de l'extérieur du module (module.h)	- constantes - déclarations de types - déclarations de variables - déclarations de prototypes de fonctions (A)
Données locales au module, inaccessibles de l'extérieur du module (module.cpp)	- constantes - déclarations de types - déclarations de variables - définitions des fonctions dont le prototype a été donné en (A) ci-dessus - définitions de fonctions locales au module

Figure 16 La notion de module.

L'implémentation de la notion de module varie d'un langage de programmation à l'autre. En Turbo Pascal, le module est mémorisé dans un fichier ; les mots-clés *interface* et *implementation* délimitent les deux parties visible et invisible du module. L'utilisateur référence le module en donnant son nom : uses module. En C, la partie interface est décrite dans un fichier à part *module.h* appelé fichier d'en-tête. La partie données locales est mémorisée dans un autre fichier *module.cpp* qui référence la partie interface en incluant *module.h*. De même, le programme utilisateur référence l'interface en faisant une inclusion de *module.h*, ce qui définit pour lui, la partie interface. Cette notion de module est aussi référencée sous le terme d'unité de compilation ou de compilation séparée. Chaque unité de compilation connaît grâce aux fichiers d'en-tête, le type et les prototypes des fonctions définies dans une autre unité de compilation.

1.4.2 Exemple : module de simulation d'écran graphique

On veut faire une simulation de dessins en mode graphique. Pour cela, on définit un module qui a l'interface suivante :

- *void initialiserEcran (int nl, int nc) ;* initialise un espace mémoire de simulation de l'écran de nl lignes sur nc colonnes numérotées de 0 à nl-1, et de 0 à nc-1 ; le crayon de couleur noire est positionné au milieu de l'écran.
- *void crayonEn (int nl, int nc) ;* positionne le crayon en (nl, nc).
- *void couleurCrayon (int c) ;* définit la couleur du crayon (de 0 à 15 par exemple).
- *void ecrirePixel (int nl, int nc) ;* écrit au point (nl, nc) un pixel de la couleur du crayon (en fait écrit un caractère dépendant de la couleur du pixel).
- *void avancer (int d, int n) ;* avance de n pixels dans la direction d ; 4 directions sont retenues : gauche, haut, droite, bas.

• *void rectangle (int xcsg, int ycsg, int xcid, int ycid) ;* trace un rectangle de la couleur du crayon, de cordonnées (xcsg, ycsg) pour le coin supérieur gauche et (xcid, ycid) pour le coin inférieur droit.
• *void ecrireMessage (int nl, int nc, char* message) ;* écrit message en (nl, nc).
• *void afficherEcran() ;* affiche l'écran.
• *void effacerEcran() ;* efface l'écran.
• *void detruireEcran() ;* détruit l'écran (libère l'espace alloué).
• *void sauverEcran (char* nom) ;* sauve l'écran dans le fichier nom.

La partie interface définit également les couleurs utilisables (NOIR, BLANC) et les directions de déplacement du crayon (GAUCHE, HAUT, DROITE, BAS). Aucune variable ne figure dans la partie interface. L'utilisateur ne sait pas comment son écran est mémorisé. L'écran est, pour lui, un **type abstrait de données** (TAD).

1.4.2.a Le fichier d'en-tête de l'écran graphique

Le fichier d'en-tête décrit les objets visibles pour les utilisateurs du module (constantes NOIR et BLANC, directions de déplacement, prototypes des fonctions).

```
/* ecran.h  fichier d'en-tête pour le module ecran.cpp */

#ifndef ECRAN_H
#define ECRAN_H

#define NOIR    0
#define BLANC   15

#define GAUCHE  1
#define HAUT    2
#define DROITE  3
#define BAS     4

void initialiserEcran (int nl, int nc);
void crayonEn         (int nl, int nc);
void couleurCrayon    (int c);
void ecrirePixel      (int nl, int nc);
void avancer          (int d,  int lg);
void rectangle        (int xcsg, int ycsg, int xcid, int ycid);
void ecrireMessage    (int nl, int nc, char* message);
void afficherEcran    ();
void effacerEcran     ();
void detruireEcran    ();
void sauverEcran      (char* nom);

#endif
```

1.4.2.b Le module écran graphique

Comme l'indique la Figure 16, *ecran.cpp* contient les données locales, et les définitions des fonctions déclarées dans la partie interface. Les données globales (externes aux fonctions) étant *static* ne peuvent être référencées de l'extérieur du module. Ces données sont locales au fichier *ecran.cpp*.

```
/* ecran.cpp  simulation d'écran graphique */

#include <stdio.h>       // printf, FILE, fopen, fprintf
#include <stdlib.h>      // malloc, free, exit
#include <string.h>      // strlen
#include "ecran.h"

// données locales au fichier ecran.cpp,
// inaccessibles pour l'utilisateur du module.
// static = locales au fichier pour les variables externes aux fonctions
static char* ecran;      // pointeur sur le début de l'écran
static int   nbLig;      // nombre de lignes de l'écran
static int   nbCol;      // nombre de colonnes de l'écran
static int   ncc;        // numéro de colonne du crayon
static int   nlc;        // numéro de ligne   du crayon
static int   couleur;    // couleur du crayon

// l'écran est un tableau de caractères ecran alloué dynamiquement
// de nl lignes sur nc colonnes et mis à blanc
void initialiserEcran (int nl, int nc) {
  nbLig = nl;
  nbCol = nc;
  ecran = (char*) malloc (nbLig * nbCol * sizeof(char));
  effacerEcran ();
}

// le crayon est mis en (nl, nc)
void crayonEn (int nl, int nc) {
  nlc = nl;
  ncc = nc;
}

// la couleur du dessin est c
void couleurCrayon (int c) {
  if (c>15) c = c % 16;
  couleur = c;
}

// écrire un caractère en fonction de la couleur en (nl, nc)
void ecrirePixel (int nl, int nc) {
  static char* tabCoul = "*123456789ABCDE.";
  if ( (nl>=0) && (nl<nbLig) && (nc>=0) && (nc<nbCol) )
    ecran [nl*nbCol+nc] = tabCoul [couleur];
}

// avancer dans la direction d de lg pixels
void avancer (int d, int n) {

  switch (d) {
  case DROITE:
    for (int i=ncc; i<ncc+n; i++) ecrirePixel (nlc, i);
    ncc += n-1;
    break;
  case HAUT:
    for (int i=nlc; i>nlc-n; i--) ecrirePixel (i, ncc);
    nlc += -n+1;
    break;
  case GAUCHE:
    for (int i=ncc; i>ncc-n; i--) ecrirePixel (nlc, i);
    ncc += -n+1;
    break;
```

```
  case BAS:
    for (int i=nlc; i<nlc+n; i++) ecrirePixel (i, ncc);
    nlc += n-1;
    break;
  }  // switch
}

// tracer un rectangle défini par 2 points csg et cid
void rectangle (int xcsg, int ycsg, int xcid, int ycid) {
  int longueur = xcid-xcsg+1;
  int largeur  = ycid-ycsg+1;

  crayonEn (ycsg,   xcsg);
  avancer  (BAS,    largeur);
  avancer  (DROITE, longueur);
  avancer  (HAUT,   largeur);
  avancer  (GAUCHE, longueur);
}

// écrire un message à partir de (nl, nc)
void ecrireMessage (int nl, int nc, char* message) {
  for (int i=0; i<strlen(message); i++) {
    if ( (nl>=0)   && (nl<nbLig) && (nc>=0) && (nc<nbCol) ) {
      ecran [nl*nbCol+nc] = message[i];
      nc++;
    }
  }
}

// afficher le dessin
void afficherEcran () {
  for (int i=0; i<nbLig; i++) {
    for (int j=0; j<nbCol; j++) printf ("%c", ecran [i*nbCol+j]);
    printf ("\n");
  }
  printf ("\n");
}

// mettre l'écran à blanc
void effacerEcran () {
  for (int i=0; i<nbLig; i++) {
    for (int j=0; j<nbCol; j++) ecran[i*nbCol+j] = ' ';
  }
  couleurCrayon (NOIR);            // par défaut
  crayonEn (nbLig/2, nbCol/2);     // milieu de l'écran
}

// rendre au système d'exploitation, l'espace alloué par
// malloc() dans initialiserEcran()
void detruireEcran () {
  free (ecran);
}

// sauver l'écran dans le fichier nomFS
void sauverEcran (char* nomFS) {
  FILE* fs = fopen (nomFS, "w");
  if (fs==NULL) { perror ("sauverEcran"); exit (1); };
  for (int i=0; i<nbLig; i++) {
    for (int j=0; j<nbCol; j++) fprintf (fs, "%c", ecran [i*nbCol+j]);
    fprintf (fs, "\n");
  }
  fprintf (fs, "\n");
  fclose (fs);
}
```

1.4.2.c Le programme d'application de l'écran graphique

Ce programme principal utilise le module *écran* pour dessiner une maison. Bien sûr, il faudrait augmenter la résolution pour avoir un dessin plus fin.

```
/* ppecran.cpp   programme principal ecran */

#include <stdio.h>
#include "ecran.h"

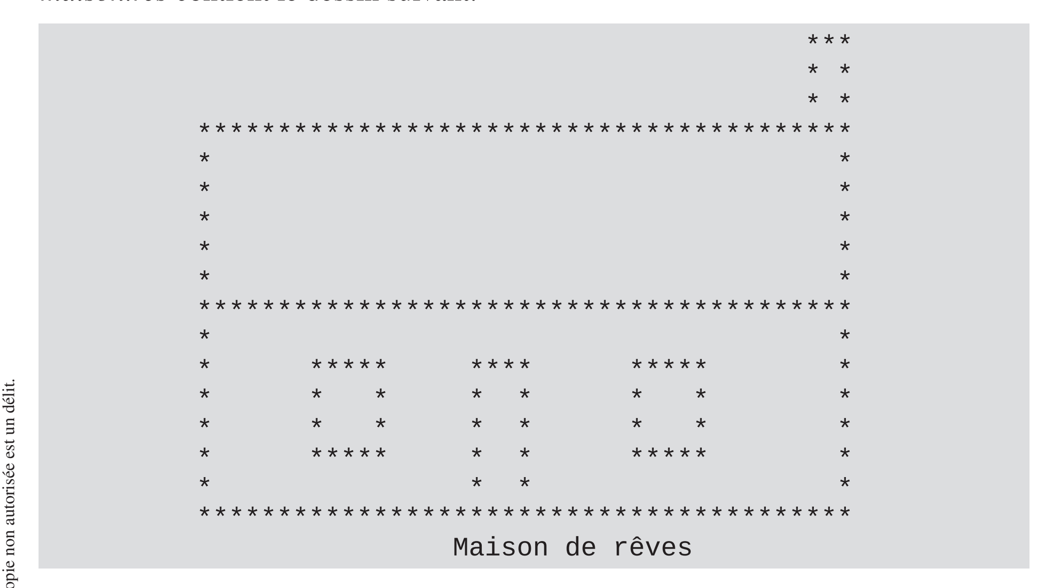
void main () {
  initialiserEcran (20, 50);

  rectangle ( 3, 10, 43, 17);  // maison
  rectangle ( 3,  4, 43, 10);  // toiture
  rectangle (20, 12, 23, 17);  // porte
  rectangle (41,  1, 43,  4);  // cheminée
  rectangle (10, 12, 14, 15);  // fenêtre gauche
  rectangle (30, 12, 34, 15);  // fenêtre droite
  ecrireMessage (19, 15, "Maison de rêves");

  afficherEcran ();
  sauverEcran   ("maison.res");
  detruireEcran ();
}
```

1.4.2.d Le résultat de l'exécution du test du module écran

Après exécution du programme d'application *ppecran.cpp* précédent, le fichier *maison.res* contient le dessin suivant.



```
                                         ***
                                         * *
                                         * *
   *****************************************
   *                                       *
   *                                       *
   *                                       *
   *                                       *
   *                                       *
   *****************************************
   *                                       *
   *      *****     ****      *****        *
   *      *   *     *  *      *   *        *
   *      *   *     *  *      *   *        *
   *      *****     *  *      *****        *
   *                *  *                   *
   *****************************************

               Maison de rêves
```

Figure 17 Dessin d'une maison avec le module ecran.

La notion de classe en programmation objet correspond à l'extension de la notion de module. Une classe comprend des objets (données propres) et des fonctions appelées méthodes qui gèrent ses objets. Sur l'exemple, il n'y a qu'un seul écran ; si on

veut pouvoir gérer plusieurs écrans, il faut passer en paramètre de chaque fonction un pointeur d'écran : un pointeur sur une structure contenant les données spécifiques de chaque écran. En programmation (orientée) objet, on définirait une classe *ecran*, chaque élément de cette classe ayant ses propres données. D'autres mécanismes plus spécifiques de la programmation objet ne sont pas abordés ici comme l'héritage ou les méthodes virtuelles.

L'exemple donné pourrait être complété en définissant d'autres fonctions mises à la disposition de l'utilisateur. On pourrait définir des fonctions de tracé de figures géométriques (cercles, carrés, ellipses, etc.). Les directions de déplacement devraient être quelconques ; on devrait pouvoir avancer de n pas suivant un angle donné. On devrait pouvoir tourner à droite ou à gauche d'un angle donné. On pourrait refaire de cette façon le langage Logo qui facilite le dessin sur écran et l'apprentissage de la programmation.

Exercice 4 - Spirale rectangulaire (récursive)

En utilisant le module *ecran* défini précédemment, écrire la fonction *récursive void spirale (int n, int lgMax) ;* qui trace la spirale de la Figure 18 ; n est la longueur du segment ; lgMax est la longueur du plus grand segment à tracer.

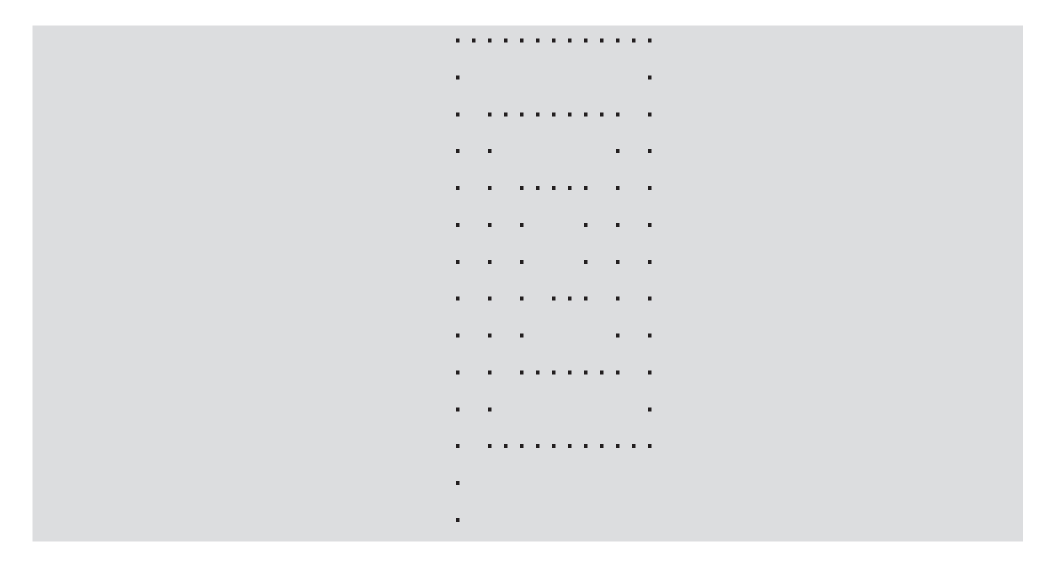


Figure 18 Dessin d'une spirale rectangulaire.

Exercice 5 - Module de gestion de piles d'entiers (allocation contiguë)

Une pile d'entiers est une structure de données contenant des entiers qui sont gérés en ajoutant et en retirant des valeurs en sommet de pile uniquement. On utilise une allocation contiguë : un tableau d'entiers géré comme une pile ; c'est-à-dire où les ajouts et les retraits sont effectués au sommet de la pile uniquement. Le type *Pile*

décrit une structure contenant une variable *max* indiquant le nombre maximum d'éléments dans la pile, une variable *nb* repérant le dernier occupé de la pile (sommet de pile) et le tableau *element* des éléments de la pile. Les fonctions à appliquer à la pile consistent à initialiser une pile en allouant dynamiquement le tableau des *max* entiers, à tester si la pile est vide ou non, à ajouter une valeur en sommet de pile s'il reste de la place, à extraire une valeur du sommet de pile si la pile n'est pas vide, à lister pour vérification toutes les valeurs de la pile, et à détruire la pile. Le fichier d'en-tête *pile.h* est le suivant :

```
/* pile.h  version avec allocation dynamique du tableau */

#ifndef PILE_H
#define PILE_H

typedef struct {
  int  max;         // nombre maximum d'éléments dans la pile
  int  nb;          // repère le dernier occupé de element
  int* element;     // le tableau des éléments de la pile
} Pile;

Pile*  creerPile    (int   max);
int    pileVide     (Pile* p);
void   empiler      (Pile* p, int  valeur);
int    depiler      (Pile* p, int* valeur);
void   listerPile   (Pile* p);
void   detruirePile (Pile* p);

#endif
```

La fonction *int depiler (Pile* p, int* valeur) ;*

• fournit 0 (faux) si la pile est vide, 1 (vrai) sinon.

• met dans l'entier pointé par valeur, et la supprime de la pile, la valeur en sommet de pile (cas de la pile non vide).

Écrire le corps du module *pile.cpp* et un programme de test *pppile.cpp* (programme principal des piles) contenant un menu permettant de vérifier tous les cas. Le menu est proposé ci-dessous.

```
GESTION D'UNE PILE

0 - Fin
1 - Création    de la pile
2 - La pile est-elle vide ?
3 - Insertion   dans la pile
4 - Retrait     de   la pile
5 - Listage     de   la pile

Votre choix ?
```

Exercice 6 - Module de gestion de nombres complexes

Un nombre complexe se compose d'une partie réelle et d'une partie imaginaire.

Les fonctions que l'on peut réaliser sur les complexes sont données ci-dessous :

- fonction *(crC)* permettant la création d'un nombre complexe à partir de 2 réels,
- fonction *(crCP)* permettant la création d'un nombre complexe à partir de son module, et de son argument en radians entre -π et +π,
- fonctions *(partReelC, partImagC, moduleC, argumentC)* délivrant la partie réelle, la partie imaginaire, le module ou l'argument d'un nombre complexe,
- fonctions *(ecritureC, ecritureCP)* faisant l'écriture des 2 composantes d'un nombre complexe sous la forme (x + y i) comme par exemple (2 + 1.5 i), ou sous la forme en polaire (2.5, 0.64),
- fonctions *(opposeC, conjugueC, inverseC, puissanceC)* délivrant le complexe opposé, le conjugué, l'inverse ou une puissance entière d'un nombre complexe,
- fonctions *(additionC, soustractionC, multiplicationC, divisionC)* faisant l'addition, la soustraction, la multiplication ou la division de deux nombres complexes.

Le fichier d'en-tête *complex.h* décrivant l'interface du module est le suivant. Le type *Complex* est décrit comme une structure de deux réels *partReel* et *partImag*.

```
/* complex.h */

#ifndef COMPLEX_H
#define COMPLEX_H

#define M_PI 3.1415926535

typedef struct {
  double partReel;       // partie réelle
  double partImag;       // partie imaginaire
} Complex;

// les constructeurs
Complex  crC            (double partReel, double partImag);
Complex  crCP           (double module,   double argument);

double   partReelC      (Complex z);
double   partImagC      (Complex z);
double   moduleC        (Complex z);
double   argumentC      (Complex z);

void     ecritureC      (Complex z);
void     ecritureCP     (Complex z);

Complex  opposeC        (Complex z);
Complex  conjugueC      (Complex z);
Complex  inverseC       (Complex z);
Complex  puissanceC     (Complex z, int n);

Complex  additionC      (Complex z1, Complex z2);
Complex  soustractionC  (Complex z1, Complex z2);
```

```
Complex  multiplicationC (Complex z1, Complex z2);
Complex  divisionC       (Complex z1, Complex z2);

#endif
```

Écrire dans le fichier *complex.cpp*, les fonctions du module déclarées dans *complex.h*.

Écrire un programme principal de test des différentes fonctions du module.

Exemples de résultats (la dernière colonne donne les résultats en polaire) :

```
c1 =          (  2.00 +  1.50 i )  (  2.50,  0.64)
c2 =          ( -2.00 +  1.75 i )  (  2.66,  2.42)
c3 = c1 + c2  (  0.00 +  3.25 i )  (  3.25,  1.57)
c4 = c1 - c2  (  4.00 + -0.25 i )  (  4.01, -0.06)
c5 = c1 * c2  ( -6.63 +  0.50 i )  (  6.64,  3.07)
c6 = c1 / c2  ( -0.19 + -0.92 i )  (  0.94, -1.78)
c7 = c1 ** 3  ( -5.50 + 14.63 i )  ( 15.62,  1.93)


p1 =          (  0.71 +  0.71 i )  (  1.00,  0.79)
p2 =          (  0.00 +  1.00 i )  (  1.00,  1.57)
p3 = p1 + p2  (  0.71 +  1.71 i )  (  1.85,  1.18)
p4 = p1 - p2  (  0.71 + -0.29 i )  (  0.77, -0.39)
p5 = p1 * p2  ( -0.71 +  0.71 i )  (  1.00,  2.36)
p6 = p1 / p2  (  0.71 + -0.71 i )  (  1.00, -0.79)
p7 = p1 ** 3  ( -0.71 +  0.71 i )  (  1.00,  2.36)
```

p1 a pour module 1 et pour argument : $\pi/4 = 0.79$

p2 a pour module 1 et pour argument : $\pi/2 = 1.57$

1.5 POINTEURS DE FONCTIONS

La plupart des langages de programmation autorise le passage en paramètre d'une fonction définie par ses propres paramètres et son type de retour. Le tracé d'une courbe peut se faire en passant la fonction en paramètre de la fonction de tracé de dessin.

Les exemples ci-dessous définissent plusieurs courbes ayant deux paramètres entiers et fournissant une valeur entière.

```
// fnparam.cpp  passage en paramètre de fonctions

#include <stdio.h>
```

```
// les différentes fonctions de deux paramètres entiers

int fsom (int n1, int n2) {
  return n1 + n2;
}

int fdif (int n1, int n2) {
  return n1 - n2;
}

int fmult (int n1, int n2) {
  return n1 * n2;
}

int fdiv (int n1, int n2) {
  return n1 / n2;
}
```

La fonction *calculer()* accepte un premier paramètre de type pointeur de fonction ayant deux entiers en paramètre et délivrant un entier.

```
// la fonction calculer ayant un paramètre
// de type fonction de deux entiers fournissant un entier
int calculer ( int (*f) (int, int), int v1, int v2) {
  int resu = f (v1,v2);
  return resu;
}
```

calculer (fsom, 10, 20) : exécuter la fonction *calculer()* avec comme premier paramètre la fonction *fsom*.

```
void main () {
  printf ("fsom  %d\n", calculer (fsom,  10, 20) );
  printf ("fdif  %d\n", calculer (fdif,  10, 20) );
  printf ("fmult %d\n", calculer (fmult, 10, 20) );
  printf ("fdiv  %d\n", calculer (fdiv,  10, 20) );

  // on peut utiliser une variable
  // de type fonction de deux entiers fournissant un entier
  // et lui affecter une valeur
  int (*f) (int, int) = fsom;
  printf ("\nf     %d\n", calculer (f,  10, 20) );
}
```

1.6 RÉSUMÉ

Certains problèmes peuvent être résolus plus logiquement en utilisant la récursivité. Les programmes sont plus compacts, plus faciles à écrire et à comprendre. Son usage est naturel quand les *structures de données sont définies récursivement,* ou quand le problème à traiter peut se *décomposer en deux* ou plus sous-problèmes identiques au problème initial mais avec des valeurs de paramètres différentes. Refuser la récursivité dans ce dernier cas oblige l'utilisateur à gérer lui-même une pile des différentes valeurs des variables, ce que le système fait automatiquement lors de l'utilisation de la récursivité.

L'allocation dynamique permet de demander de la mémoire centrale au système d'exploitation au fil des besoins. L'allocation dynamique de mémoire pour l'écran graphique (voir *initialiserEcran()*) permet d'allouer une zone mémoire contiguë en fonction des dimensions voulues de l'écran. En allocation statique (déclaration d'un tableau à deux dimensions), il aurait fallu figer lors de la déclaration les deux dimensions de l'écran pour toutes les applications. La structure de données des relations entre personnes de la Figure 14 illustre bien la nécessité de pouvoir allouer de nouvelles zones dynamiquement, en cours d'exécution du programme, pour y enregistrer les caractéristiques d'une nouvelle personne. Les différentes zones allouées sont reliées entre elles à l'aide de pointeurs.

Comme dans toute réalisation humaine complexe, il convient d'être méthodique et de décomposer le problème en sous-problèmes de moindre difficulté. La construction d'une voiture se fait en assemblant des modules (moteur, boîte de vitesses, etc.) qui doivent respecter des normes d'interface très précises. Si les normes sont respectées, on peut facilement remplacer le moteur par un autre moteur plus performant. Il en va de même en programmation. Une application doit être découpée en modules qui communiquent par des interfaces définies sous forme de prototypes de fonctions. On pourra toujours remplacer un module par un module plus performant dès lors que l'interface ne change pas.

Chapitre 2

Les listes

2.1 LISTES SIMPLES : DÉFINITION

Une liste est un ensemble d'objets de même type constituant les éléments de la liste. Les éléments sont chaînés entre eux et on peut facilement ajouter ou extraire un ou plusieurs éléments. Une liste simple est une structure de données telle que chaque élément contient :

• des informations caractéristiques de l'application (les caractéristiques d'une personne par exemple),

• un pointeur vers un autre élément ou une marque de fin s'il n'y a pas d'élément successeur.

Exemple : une liste d'attente chez un médecin.

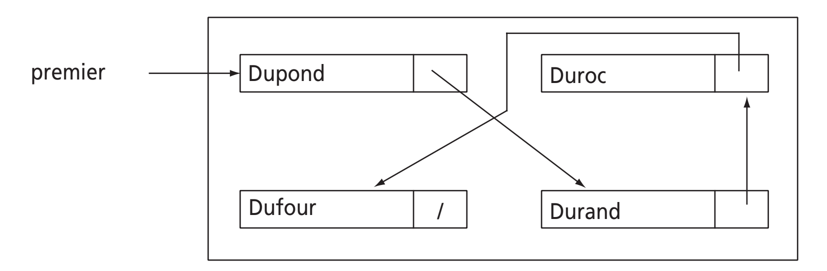


Figure 19 Une liste de clients.

La Figure 19 peut représenter une salle d'attente où les clients sont dispersés dans toute la salle. Cependant, il y a un ordre de passage correspondant à l'ordre

d'arrivée. L'arrivée d'un nouveau client n'entraîne pas de déplacement pour les clients déjà présents. Il doit occuper un des sièges libres.

Le schéma peut aussi représenter les informations sur les clients dispersées en mémoire centrale d'ordinateur mais reliées entre elles par un chaînage. La mémorisation d'un nouveau client se fait par demande au système d'exploitation d'une zone mémoire libre pour y loger les caractéristiques du nouveau client. Les informations sur les autres clients n'ont pas à être déplacées.

En salle d'attente, on pourrait aussi imaginer un banc d'attente avec décalage des clients à chaque passage chez le médecin. Cela correspondrait à une gestion en tableau en mémoire centrale avec décalage des informations.

Les listes permettent une gestion sans déplacement en mémoire quand l'information est volatile, c'est-à-dire quand il y a de fréquents ajouts et retraits d'éléments. Une gestion en tableau est difficile quand des éléments doivent être ajoutés ou retirés en milieu de tableau. Si les éléments sont toujours ajoutés ou retirés en début ou en fin de liste, une structure en tableau peut être envisagée.

2.2 REPRÉSENTATION EN MÉMOIRE DES LISTES

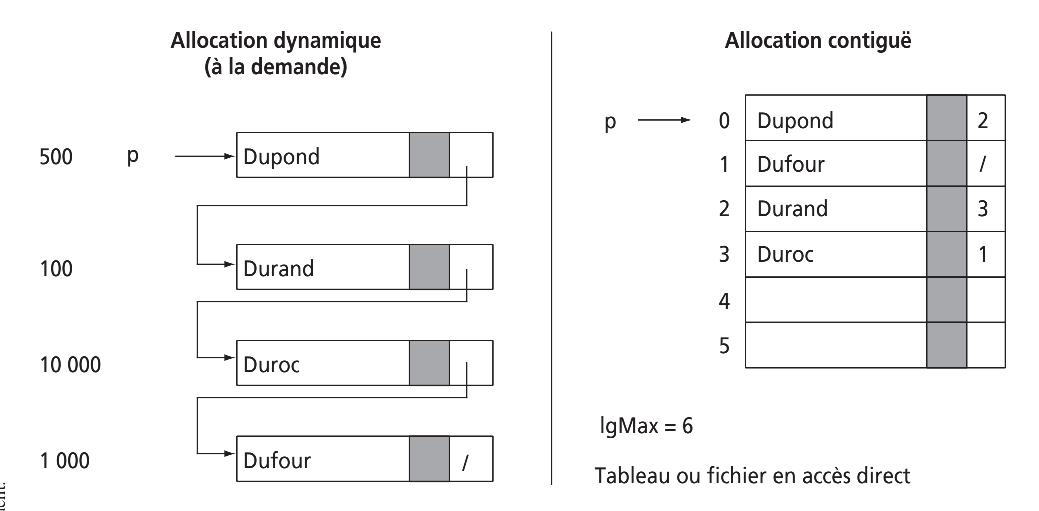


Figure 20 Mémorisation des listes (dynamique ou contiguë).

En allocation dynamique à la demande, les différents éléments sont dispersés en mémoire centrale. L'espace est réservé au fur et à mesure des créations d'éléments. La seule limite est la taille de la mémoire centrale de l'ordinateur ou l'espace mémoire alloué au processus. Sur l'exemple de la Figure 20, la structure de Dupond se trouve à l'adresse 500, celle de Durand à l'adresse 100. Ainsi, le suivant de Dupond se trouve à l'adresse 100. Cette adresse est un pointeur sur l'élément suivant.

Le schéma de l'allocation contiguë représente une allocation où l'espace est réservé sur des cases consécutives avec une dimension maximale (lgMax sur le schéma) à préciser lors de cette réservation. L'allocation dynamique à la demande présente beaucoup plus de souplesse puisqu'il n'y a pas de limite à fixer. Le fait de devoir fixer une limite est gênant pour une application. Si la limite est trop basse, on ne peut résoudre les problèmes ayant de nombreuses valeurs, ou alors, il faut réallouer une zone plus grande et déplacer les informations. Si la limite est trop grande, l'espace est alloué inutilement. Le schéma de l'allocation contiguë peut représenter un tableau en mémoire centrale ou un fichier en accès direct sur disque.

Les déclarations concernant les allocations dynamiques à la demande et contiguës sont données ci-dessous. Il faut remarquer que l'allocation contiguë peut être faite de manière statique (déclaration d'un tableau) ou dynamique par *malloc()* en début ou en cours de programme.

Allocation dynamique
(à la demande)

```
typedef char ch15 [16];
typedef
  struct element* PElement;

typedef struct element {
  ch15     nom;
  PElement suivant;
} Element;
```

Allocation contiguë (statique)

```
#define lgMax    6
#define NULLE    -1
typedef int PElement;

typedef struct {
  ch15     nom;
  PElement suivant;
} Element;
```

soit un tableau de structures

```
Element tab [lgMax];
```

soit un fichier en accès direct

```
FILE* f;
```

2.3 MODULE DE GESTION DES LISTES

Un **élément** de liste contient toujours un pointeur sur l'élément suivant ou une marque indiquant qu'il n'y a plus de suivant (marque NULL en C). Les autres caractéristiques dépendent de l'application. Pour que le module de gestion des listes soit le plus général possible, il faut bien séparer ce qui est spécifique des listes de ce qui est caractéristique des applications. Ci-dessous, les informations sont regroupées dans une structure (un objet) et repérées par un pointeur de type Objet* (soit void*) appelé référence. Cette référence peut contenir l'adresse de n'importe quel objet.

La liste peut être représentée par un pointeur sur le premier élément de la liste. On peut aussi regrouper quelques caractéristiques de la liste dans une structure de type tête de liste qui contient par exemple un pointeur sur le premier élément et un pointeur sur le dernier élément de façon à faciliter les insertions en tête et en fin de liste. Un pointeur courant est également ajouté pour faciliter le parcours des listes. Le type **Liste** est ainsi défini comme une structure contenant trois pointeurs d'éléments : un

pointeur sur le premier élément de la liste, un pointeur sur le dernier élément, et un pointeur qui repère l'élément courant à traiter. Le nombre d'éléments est également inséré dans la tête de liste, ainsi que le type de la liste (non ordonnée, ordonnée croissante ou ordonnée décroissante). Deux fonctions dépendant des objets traités et permettant de fournir la chaîne de caractères à écrire pour chaque objet ou de comparer deux objets complètent cette structure de type Liste. La fonction de comparaison fournit 0 si les deux objets sont égaux, une valeur inférieure à 0 si le premier objet est inférieur au deuxième, et une valeur supérieure à 0 sinon. Ces deux fonctions sont des paramètres de la liste et varient d'une application à l'autre.

Les déclarations correspondantes pour l'allocation dynamique à la demande sont les suivantes :

```
typedef void Objet;

// un élément de la liste
typedef struct element {
  Objet*          reference;   // référence un objet (de l'application)
  struct element* suivant;     // élément suivant de la liste
} Element;

// le type Liste
typedef struct {
  Element* premier; // premier élément de la liste
  Element* dernier; // dernier élément de la liste
  Element* courant; // élément en cours de traitement (parcours de liste)
  int      nbElt;   // nombre d'éléments dans la liste
  int      type;    // 0:non ordonné, 1:croissant, 2:décroissant
  char*    (*toString) (Objet*);
  int      (*comparer) (Objet*, Objet*);
} Liste;
```

Une liste ne contenant aucun élément a ses trois pointeurs à NULL pour indiquer qu'il n'y a ni premier, ni dernier, ni élément courant.

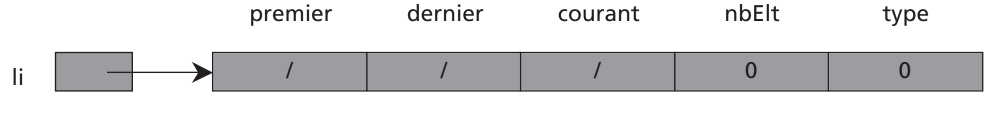


Figure 21 Une liste vide.

La fonction d'initialisation de la liste pointée par li est donnée ci-dessous. li est l'adresse d'une structure de type *Liste ;* li est donc de type *Liste**. La barre oblique / représente NULL sur la Figure 21.

li->premier indique le champ premier de la structure pointée par li.

Par défaut, l'objet référencé par l'élément de liste est une chaîne de caractères. Les deux fonctions définies ci-dessous permettent d'initialiser par défaut les paramètres fonctions de la tête de liste.

```
// comparer deux chaînes de caractères
// fournit <0 si ch1 < ch2; 0 si ch1=ch2; >0 sinon
static int comparerCar (Objet* objet1, Objet* objet2) {
  return strcmp ((char*) objet1, (char*) objet1);
}

static char* toChar (Objet* objet) {
  return (char*) objet;
}

// initialiser la liste pointée par li (cas général)
void initListe (Liste* li, int type, char* (*toString) (Objet*),
                                      int (*comparer) (Objet*, Objet*)) {
  li->premier  = NULL;
  li->dernier  = NULL;
  li->courant  = NULL;
  li->nbElt    = 0;
  li->type     = type;
  li->toString = toString;
  li->comparer = comparer;
}

// initialisation par défaut
void initListe (Liste* li) {
  initListe (li, NONORDONNE, toChar, comparerCar);
}
```

Les fonctions suivantes créent et initialisent la tête de liste, et fournissent un pointeur sur la tête de liste créée.

```
Liste* creerListe (int type, char* (*toString) (Objet*),
                                      int (*comparer) (Objet*, Objet*)) {
  Liste* li = new Liste();
  initListe (li, type, toString, comparer);
  return li;
}

Liste* creerListe (int type) {
  return creerListe (type, toChar, comparerCar);
}

Liste* creerListe () {
  return creerListe (NONORDONNE, toChar, comparerCar);
}
```

Pour savoir si une liste est vide, il suffit de tester si son nombre d'élément vaut 0.

```
// la liste est-elle vide ?
booleen  listeVide (Liste* li) {
  return li->nbElt == 0;
}
```

La fonction *nbElement()* fournit le nombre d'éléments dans la liste li :

```
// fournir le nombre d'éléments dans la liste
int nbElement (Liste* li) {
  return li->nbElt;
}
```

2.3.1 Création d'un élément de liste (fonction locale au module sur les listes)

La liste contient des éléments ; chaque élément référence un objet spécifique de l'application.

```
// créer un élément de liste
static Element* creerElement () {
  return new Element();
}
```

2.3.2 Ajout d'un objet

2.3.2.a Ajout en tête de liste

Avant l'appel de cette fonction d'ajout, l'objet à insérer et pointé par objet a été alloué et rempli avec les informations spécifiques de l'application. Un élément de liste est créé et son champ *reference* repère l'objet à insérer.

```
// insérer objet en tête de la liste li
// l'objet est repéré par le champ reference de l'élément de la liste
void insererEnTeteDeListe (Liste* li, Objet* objet) {
  Element* nouveau   = creerElement();
  nouveau->reference = objet;
  nouveau->suivant   = li->premier;
  li->premier        = nouveau;
  if (li->dernier == NULL) li->dernier = nouveau;
  li->nbElt++;
}
```

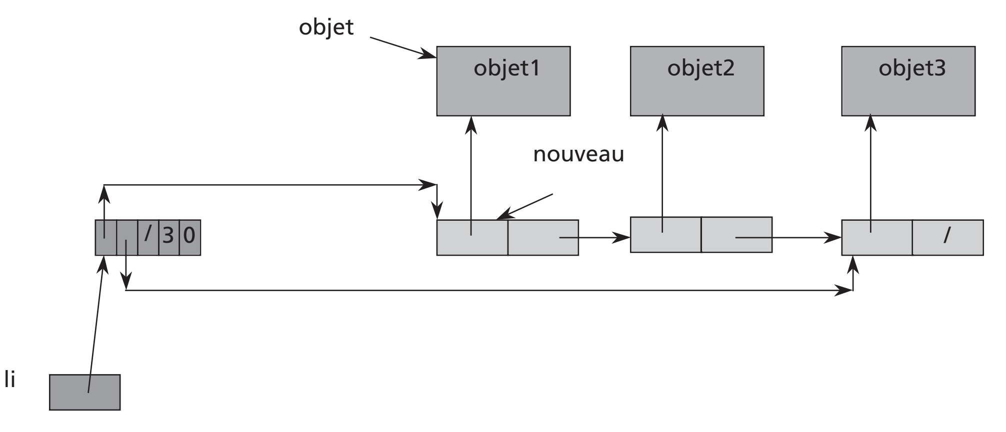


Figure 22 Insertion de objet en tête de la liste li de type 0 (non ordonnée). Après insertion, la liste contient 3 éléments.

2.3.2.b Ajout après l'élément précédent (fonction locale au module)

L'élément doit être ajouté à une place particulière après l'élément *precedent*. Il faut d'abord trouver l'élément *precedent* avant d'appeler la fonction *insererApres()*.

La procédure présentée ci-dessous insère dans la liste pointée par *li*, après l'élément pointé par *precedent*, l'élément pointé par nouveau qui référence *objet*. Si *precedent* est NULL, l'insertion se fait en tête de liste.

```
// insérer dans la liste li, objet après precedent
// si precedent est NULL, insérer en tête de liste
static void insererApres (Liste* li, Element* precedent, Objet* objet) {
  if (precedent == NULL) {
    insererEnTeteDeListe (li, objet);
  } else {
    Element* nouveau   = creerElement();
    nouveau->reference = objet;
    nouveau->suivant   = precedent->suivant;
    precedent->suivant = nouveau;
    if (precedent == li->dernier) li->dernier = nouveau;
    li->nbElt++;
  }
}
```

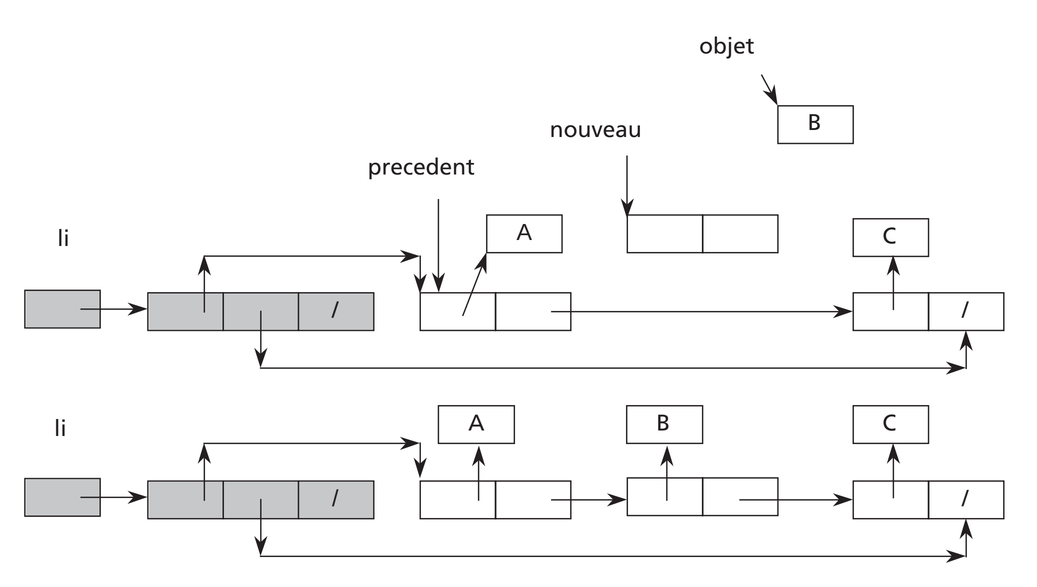


Figure 23 Insertion de nouveau (référençant objet) dans la liste li, après l'élément pointé par precedent.

2.3.2.c Ajout en fin de liste

L'ajout en fin de liste se fait en tête de liste si la liste est vide ou après le dernier élément si la liste contient déjà un élément. La fonction précédente peut donc être utilisée pour définir cette fonction. Le pointeur sur le dernier élément conservé dans la tête de liste permet une insertion en fin de liste rapide sans parcours de la liste pour se positionner sur le dernier élément. Si la liste est vide, li->dernier vaut NULL.

```
// insérer un objet en fin de la liste li
void insererEnFinDeListe (Liste* li, Objet* objet) {
  insererApres (li, li->dernier, objet);
}
```

2.3.3 Les fonctions de parcours de liste

Les fonctions suivantes permettent à l'utilisateur du module *liste* de parcourir une liste en faisant abstraction des structures de données sous-jacentes. Ces fonctions s'apparentent à celles utilisées pour parcourir séquentiellement les fichiers. L'utilisateur a besoin de se positionner en début de liste, de demander l'objet suivant de la liste, et de savoir s'il a atteint la fin de la liste. L'utilisateur n'accède pas aux champs de la structure de la tête de liste (premier, dernier, courant), ni au champ suivant des éléments.

La fonction *ouvrirListe()* permet de se positionner sur le premier élément de la liste *li*.

```
// se positionner sur le premier élément de la liste li
void ouvrirListe (Liste* li) {
  li->courant = li->premier;
}
```

La fonction booléenne *finListe()* indique si on a atteint la fin de la liste *li* ouverte préalablement par `ouvrirListe()`.

```
// a-t-on atteint la fin de la liste li ?
booleen finListe (Liste* li) {
  return li->courant==NULL;
}
```

La fonction **locale** *elementCourant()* fournit un pointeur sur l'élément courant de la liste *li* et se positionne sur l'élément suivant qui devient l'élément courant.

```
// fournir un pointeur sur l'élément courant de la liste li,
// et se positionner sur le suivant qui devient le courant
static Element* elementCourant (Liste* li) {
  Element* ptc = li->courant;
  if (li->courant != NULL) {
    li->courant = li->courant->suivant;
  }
  return ptc;
}
```

La fonction *objetCourant()* fournit un pointeur sur l'objet courant de la liste *li*. Chaque appel déplace l'objet courant sur le suivant.

```
// fournir un pointeur sur l'objet courant de la liste li,
// et se positionner sur le suivant qui devient le courant
Objet* objetCourant (Liste* li) {
  Element* ptc = elementCourant (li);
  return ptc==NULL ? NULL : ptc->reference;
}
```

La fonction *listerListe(Liste* li)* effectue un parcours complet de la liste en appliquant la fonction *toString()* spécifique des objets de l'application et définie dans la tête de liste lors de la création de la liste (voir *creerListe()*).

```
void listerListe (Liste* li) {
  ouvrirListe (li);
  while (!finListe (li)) {
    Objet* objet = objetCourant (li);
    printf ("%s\n", li->toString (objet));
  }
}
```

La fonction *listerListe(Liste* li, void (*f) (Objet*))*) effectue un parcours complet de la liste en appliquant la fonction *f()* donnée en paramètre pour chacun des éléments de la liste. La fonction *f()* est spécifique de l'application. Par contre la façon de faire le parcours est, elle, indépendante de cette application.

```
// lister la liste li;
// f est une fonction passée en paramètre
// et ayant un pointeur de type quelconque.
// Ceci s'apparente aux méthodes virtuelles en PO.
void listerListe (Liste* li, void (*f) (Objet*)) {
  ouvrirListe (li);
  while (!finListe (li)) {
    Objet* objet = objetCourant (li);
    f (objet);  // appliquer la fonction f() à objet
  }
}
```

De manière assez similaire, la fonction *chercherUnObjet()* effectue un parcours de liste en comparant l'objet cherché et l'objet référencé par l'élément courant de la liste. Cette comparaison est dépendante de l'application et confiée à la fonction de comparaison de deux objets définie lors de la création de la liste (voir *creerListe()*). La fonction de comparaison retourne 0 en cas d'égalité des deux objets. *objet-Cherche* doit contenir les caractéristiques (la clé) permettant (à la fonction de comparaison) d'identifier l'objet cherché dans la liste *li*.

```
// fournir un pointeur sur l'objet "objetCherche" de la liste li;
// NULL si l'objet n'existe pas
Objet* chercherUnObjet (Liste* li, Objet* objetCherche) {
  booleen trouve = faux;
  Objet* objet;       // pointeur courant
  ouvrirListe (li);
  while (!finListe (li) && !trouve) {
    objet  = objetCourant (li);
    trouve = li->comparer (objetCherche, objet) == 0;
  }
  return trouve ? objet : NULL;
}
```

2.3.4 Retrait d'un objet

2.3.4.a Retrait en tête de liste

Il s'agit de retirer l'objet en tête de la liste pointée par li, et de fournir un pointeur sur l'objet extrait. Si la liste est vide, on ne peut retirer aucun élément, la fonction retourne NULL pour indiquer un échec.

```
// extraire l'objet en tête de la liste li
Objet* extraireEnTeteDeListe (Liste* li) {
  Element* extrait = li->premier;
  if (!listeVide(li)) {
    li->premier = li->premier->suivant;
    if (li->premier==NULL) li->dernier=NULL; // Liste devenue vide
    li->nbElt--;
  }
  return extrait != NULL ? extrait->reference : NULL;
}
```

Avant : li = (C, A, B)

Après : li = (A, B)

Figure 24 Retrait de l'objet en tête de la liste pointée par li.

2.3.4.b Retrait de l'élément qui suit l'élément précédent (fonction locale)

Pour extraire un élément d'une liste, il faut avoir un pointeur sur l'élément qui précède puisqu'après extraction, le champ suivant du précédent doit contenir un pointeur sur le suivant de l'élément à extraire. Si precedent vaut NULL, il s'agit d'une extraction en tête de liste. Si on extrait le dernier élément de la liste, il faut modifier le pointeur sur le dernier qui pointe, après extraction, sur precedent. La fonction retourne un pointeur sur l'élément extrait (qui peut par la suite être détruit, ou réinséré dans une autre liste).

```
// Extraire l'objet de li se trouvant après l'élément precedent;
// si precedent vaut NULL, on extrait le premier de la liste;
// retourne NULL si l'objet à extraire n'existe pas
static Objet* extraireApres (Liste* li, Element* precedent) {
  if (precedent == NULL) {
    return extraireEnTeteDeListe (li);
  } else {
    Element* extrait = precedent->suivant;
    if (extrait != NULL) {
      precedent->suivant = extrait->suivant;
      if (extrait == li->dernier) li->dernier = precedent;
      li->nbElt--;
    }
```

```
    return extrait != NULL ? extrait->reference : NULL;
  }
}
```

Remarque : pour extraire un élément d'une liste connaissant uniquement un pointeur sur l'élément à extraire et si ce n'est pas le dernier élément de la liste, on peut permuter l'élément à extraire et son suivant, et extraire le suivant.

2.3.4.c Retrait de l'objet en fin de liste

Pour extraire le dernier élément d'une liste, il faut connaître l'avant-dernier pour en modifier le pointeur suivant. Sauf si la liste ne contient aucun, ou un seul élément, il faut faire un parcours de la liste pour repérer le précédent du dernier.

```
// extraire l'objet en fin de la liste li
Objet* extraireEnFinDeListe (Liste* li) {
  Objet* extrait;
  if (listeVide(li)) {
    extrait = NULL;
  } else if (li->premier == li->dernier) {  // un seul élément
    extrait = extraireEnTeteDeListe (li);
  } else {
    Element* ptc = li->premier;
    while (ptc->suivant != li->dernier) ptc = ptc->suivant;
    extrait = extraireApres (li, ptc);
  }
  return extrait;
}
```

2.3.4.d Retrait d'un objet à partir de sa référence

La fonction *extraireUnObjet()* extrait un objet connaissant un pointeur sur cet objet :

```
// extraire de la liste li, l'objet pointé par objet
booleen extraireUnObjet (Liste* li, Objet* objet) {
  Element* precedent = NULL;
  Element* ptc       = NULL;

  // repère l'élément précédent
  booleen trouve = faux;
  ouvrirListe (li);
  while (!finListe (li) && !trouve) {
    precedent = ptc;
    ptc       = elementCourant (li);
    trouve = (ptc->reference == objet) ? vrai : faux;
  }
  if (!trouve) return faux;

  Objet* extrait = extraireApres (li, precedent);
  return vrai;
}
```

2.3.5 Destruction de listes

Pour détruire une liste, il faut effectuer un parcours de liste avec destruction de chaque élément. La tête de liste est réinitialisée. Il faut se positionner en début de liste, et tant qu'on n'a pas atteint la fin de la liste, il faut prendre l'élément courant et le détruire. Le pointeur sur le prochain élément est conservé dans le champ courant de la tête de liste.

```
// parcours de liste avec destruction de chaque élément
void detruireListe (Liste* li) {
  ouvrirListe (li);
  while (!finListe (li)) {
    Element* ptc = elementCourant (li);
    //free (ptc->reference); //si on veut détruire les objets de la liste
    free (ptc);
  }
  initListe (li);
}
```

2.3.6 Recopie de listes

La fonction *void recopierListe (Liste* l1, Liste* l2) ;* permet de transférer la liste l2 dans la liste l1 en réinitialisant la liste l2 qui est vide.

```
// recopie l2 dans l1 et initialise l2
void recopierListe (Liste* l1, Liste* l2) {
  detruireListe (l1);
  *l1 = *l2;       // on recopie les têtes de listes
  initListe (l2);
}
```

2.3.7 Insertion dans une liste ordonnée

L'insertion dans une liste ordonnée se fait toujours suivant le même algorithme. Cependant la comparaison de l'objet à insérer par rapport aux objets déjà dans la liste dépend de l'objet de l'application. La comparaison peut porter sur des entiers, des réels, des chaînes de caractères, ou même sur plusieurs champs (nom et prénom par exemple). La fonction **locale** *enOrdre()* suivante indique si objet1 et objet2 sont en ordre (croissant si ordreCroissant est vrai, décroissant sinon). Elle utilise la fonction *comparer()* qui fournit une valeur <0 si objet1 < objet2, égale à 0 si objet1 = objet2, et supérieure à 0 sinon.

```
// objet1 et objet2 sont-ils en ordre ?
static booleen enOrdre (Objet* objet1, Objet* objet2, booleen ordreCroissant,
                                     int (*comparer) (Objet*, Objet*)) {
  booleen ordre = comparer (objet1, objet2) < 0;
  if (!ordreCroissant) ordre = !ordre;
  return ordre;
}
```

Ainsi :

enOrdre de 10 et 20 en ordre CROISSANT	vrai
enOrdre de 10 et 20 en ordre DECROISSANT	faux
enOrdre de "Dupond" et "Duval" en ordre CROISSANT	vrai

La fonction

```
void insererEnOrdre (Liste* li, Objet* objet);
```

insère dans la liste *li*, l'objet pointé par *objet* suivant le type croissant ou décroissant de la liste défini lors de la création de la liste (voir *creerListe()*). Plusieurs cas sont à envisager. Si la liste *li* est vide, il faut insérer *objet* en tête de la liste. Si *objet* doit être inséré avant le premier élément, il s'agit également d'une insertion en tête de liste. Sinon, il faut rechercher un point d'insertion tel que *objet* et l'objet de l'élément courant de la liste soient en ordre tout en gardant un pointeur sur l'élément précédent. Si on atteint la fin de la liste sans trouver ce point d'insertion, il s'agit d'une insertion en fin de liste.

Cette fonction est **indépendante** du type des objets de l'application, le test étant reporté dans la fonction *enOrdre()*. Des exemples d'utilisation sont donnés dans les applications qui suivent.

```
// la fonction comparer
// dépend du type de l'objet inséré dans la liste
void insererEnOrdre (Liste* li, Objet* objet) {
  if (listeVide (li) ) {   // liste vide
    insererEnTeteDeListe (li, objet);
    //printf ("insertion dans liste vide\n");
  } else {
    Element* ptc = li->premier;
    if ( enOrdre (objet, ptc->reference, li->type==1, li->comparer) ) {
      // insertion avant le premier élément
      //printf ("insertion en tête de liste non vide\n");
      insererEnTeteDeListe (li, objet);
    } else {    // insertion en milieu ou fin de liste
      //printf ("insertion en milieu ou fin de liste non vide\n");
      booleen  trouve = faux;
      Element* prec   = NULL;
      while (ptc != NULL && !trouve) {
        prec   = ptc;
        ptc    = ptc->suivant;
        if (ptc!=NULL) trouve = enOrdre (objet, ptc->reference,
                                       li->type==1, li->comparer);
      }
      // insertion en milieu de liste ou fin de liste
      insererApres (li, prec, objet);
    }
  }
}
```

2.3.8 Le module de gestion de listes

Le module *liste* (voir Figure 16, page 25) facilite la gestion des listes d'objets. Il se compose d'un fichier d'en-tête *liste.h* décrivant l'interface du module et du corps *liste.cpp* du module.

2.3.8.a Le fichier d'en-tête des listes simples

Le fichier d'en-tête *liste.h* contient les définitions des types Objet et Liste, et les prototypes des fonctions du module de gestion de listes. Il doit être inclus dans chaque application traitant des listes. Objet* (équivalent à void*) indique un pointeur (sans type) sur un objet dépendant de l'application. L'utilisateur du module ne gère que des objets. Il n'a pas connaissance de la façon dont ceux-ci sont mémorisés dans la liste. Il n'utilise pas le type Element, ni les fonctions traitant un Element. Celles-ci sont locales au module liste et n'apparaissent que dans liste.cpp.

```
// liste.h

#ifndef LISTE_H
#define LISTE_H

#define faux   0
#define vrai   1
typedef int    booleen;

typedef void Objet;

#define NONORDONNE  0
#define CROISSANT   1
#define DECROISSANT 2

// un élément de la liste
typedef struct element {
  Objet*          reference;   // référence un objet (de l'application)
  struct element* suivant;     // élément suivant de la liste
} Element;

// le type Liste
typedef struct {
  Element* premier; // premier élément de la liste
  Element* dernier; // dernier élément de la liste
  Element* courant; // élément en cours de traitement (parcours de liste)
  int      nbElt;   // nombre d'éléments dans la liste
  int      type;    // 0:simple, 1:croissant, 2:décroissant
  int      (*comparer) (Objet*, Objet*);
  char*    (*toString) (Objet*);
} Liste;

void     initListe              (Liste* li, int type,
                                           char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*) );
void     initListe              (Liste* li);
Liste*   creerListe             (int type, char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*) );
Liste*   creerListe             (int type);
Liste*   creerListe             ();

booleen  listeVide              (Liste* li);
int      nbElement              (Liste* li);

void     insererEnTeteDeListe   (Liste* li, Objet* objet);
void     insererEnFinDeListe    (Liste* li, Objet* objet);
```

```
// parcours de liste
void     ouvrirListe           (Liste* li);
booleen  finListe              (Liste* li);
Objet*   objetCourant          (Liste* li);
void     listerListe           (Liste* li);
void     listerListe           (Liste* li, void (*f) (Objet*));
Objet*   chercherUnObjet       (Liste* li, Objet* objetCherche);

Objet*   extraireEnTeteDeListe (Liste* li);
Objet*   extraireEnFinDeListe  (Liste* li);
booleen  extraireUnObjet       (Liste* li, Objet* objet);

void     detruireListe         (Liste* li);
void     recopierListe         (Liste* l1, Liste* l2);


// LISTE ORDONNEE
void     insererEnOrdre        (Liste* li, Objet* objet);

#endif
```

Remarque : La fonction en paramètre *toString()* doit être définie si on utilise la fonction *listerListe (Liste*)*.

De même, la fonction en paramètre *comparer()* doit être définie si on utilise les fonctions *chercherUnObjet()* ou *insererEnOrdre()*.

2.3.8.b Le module des listes simples

Le fichier *liste.cpp* contient les corps des fonctions dont les prototypes sont donnés dans *liste.h*. Le corps de ces fonctions a été vu dans les paragraphes précédents. Le module se présente donc comme suit :

```
/* liste.cpp
   Ce module de gestion de listes est très général
   et indépendant des applications. Il gère des listes
   simples d'éléments avec tête de liste. */

#include <stdlib.h>

#include "liste.h"
```

puis ensuite, le corps des fonctions définies dans *liste.h* : *initListe()*, *creerListe()*, *listeVide()*, ..., *detruireListe()*, *recopierListe()*, *insererEnOrdre()*,

et les fonctions locales au module et déclarées *static* suivantes. Le corps de ces fonctions a été vu dans les paragraphes précédents.

```
// locales au module
static Element* creerElement   ();
static void     insererApres   (Liste* li, Element* precedent,
                                                    Objet* objet);
static Objet*   extraireApres  (Liste* li, Element* precedent);
static Element* elementCourant (Liste* li);
static booleen  enOrdre        (Objet* objet1, Objet* objet2,
                                           booleen ordreCroissant,
                                int (*comparer) (Objet*, Objet*)) ;
```

2.4 EXEMPLES D'APPLICATION

2.4.1 Le type Personne

Le type Personne définit une structure (un objet) comportant un nom et un prénom. Quelques fonctions utilisant cette structure sont définies ci-dessous. Elles permettent de créer et d'initialiser une structure (un objet) de type Personne, d'écrire les caractéristiques d'une personne et de comparer deux personnes.

Le fichier d'en-tête *mdtypes.h* contient la déclaration du type Personne :

```
/* mdtypes.h */

#ifndef MDTYPES_H
#define MDTYPES_H

typedef char ch15 [16];
typedef void Objet;

// une personne
typedef struct {
  ch15 nom;
  ch15 prenom;
} Personne;

Personne* creerPersonne    (char* nom, char* prenom);
Personne* creerPersonne    ();
void      ecrirePersonne   (Objet* objet);
char*     toStringPersonne (Objet* objet);
int       comparerPersonne (Objet* objet1, Objet* objet2);

#endif
```

Le fichier des fonctions mdtypes.cpp :

```
/* mdtypes.cpp   différents types */

#include <stdio.h>
#include <string.h>      // strcpy, strcmp
#include "mdtypes.h"

// constructeur de Personne
Personne* creerPersonne (char* nom, char* prenom) {
  Personne* p = new Personne();
  strcpy (p->nom, nom);
  strcpy (p->prenom, prenom);
  return p;
}

// lecture du nom et prénom
Personne* creerPersonne () {
  printf ("Nom de la personne à créer ? ");
  ch15 nom; scanf ("%s", nom);
  printf ("Prénom de la personne à créer ? ");
  ch15 prenom; scanf ("%s", prenom);
  Personne* nouveau = creerPersonne (nom, prenom);
  return nouveau;
}
```

```
// écrire les caractéristiques d'une personne
void ecrirePersonne (Personne* p) {
  printf ("%s %s\n", p->nom, p->prenom);
}

// fournir les caractéristiques d'une personne
char* toStringPersonne (Personne* p) {
  char* message = (char*) malloc (30);  // test à faire
  sprintf (message, "%s %s", p->nom, p->prenom);
  return message;
}

// comparer deux personnes
// fournir <0 si p1 < p2; 0 si p1=p2; >0 sinon
int comparerPersonne (Personne* p1, Personne* p2) {
  return strcmp (p1->nom, p2->nom);
}

void ecrirePersonne (Objet* objet) {
  ecrirePersonne ( (Personne*) objet);
}

char* toStringPersonne (Objet* objet) {
  return toStringPersonne ( (Personne*) objet);
}

int comparerPersonne (Objet* objet1, Objet* objet2) {
  return comparerPersonne ( (Personne*)objet1, (Personne*)objet2);
}
```

2.4.2 Liste de personnes

Il s'agit de gérer une liste de personnes. On suppose qu'il y a de nombreux départs et arrivées de personnes. L'information étant volatile, l'utilisation d'une liste permet de résoudre le problème simplement sans réservation inutile d'espace mémoire. L'inclusion du fichier d'en-tête liste.h fournit à l'utilisateur, la déclaration du type Liste et les prototypes des fonctions. L'inclusion du fichier "mdtypes.h" définit le type Personne et les fonctions correspondantes.

```
/* pplistpers.cpp   programme principal de liste de personnes
   Utilisation du module de gestion de listes;
   Application à la gestion d'une liste de personnes */

#include <stdio.h>
#include "liste.h"
#include "mdtypes.h"

typedef Liste ListePersonnes;  // un équivalent plus mnémonique
```

Le menu et le programme principal suivants permettent l'insertion d'une nouvelle personne en tête ou en fin de liste, l'extraction de la personne en tête ou en fin de liste ou d'une personne dont on fournit le nom, l'écriture de la liste des personnes, la recherche d'une personne à partir de son nom et la destruction de la liste. On peut également initialiser une liste ordonnée à partir d'un fichier dont on fournit le nom ; l'insertion peut se faire en ordre croissant ou décroissant.

```
int menu () {
  printf ("\n\nGESTION D'UNE LISTE DE PERSONNES\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Insertion   en tête de liste\n");
```

```
  printf ("2 - Insertion   en fin  de liste\n");
  printf ("3 - Retrait     en tête de liste\n");
  printf ("4 - Retrait     en fin  de liste\n");
  printf ("5 - Retrait     d'un élément à partir de son nom\n");
  printf ("6 - Parcours    de la liste\n");
  printf ("7 - Recherche   d'un élément à partir de son nom\n");
  printf ("8 - Insertion   ordonnée à partir d'un fichier\n");
  printf ("9 - Destruction de la liste\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod);
  printf ("\n");

  return cod;
}


void main () {
  Liste* lp = creerListe (0, toStringPersonne, comparerPersonne);
  booleen fini = faux;

  while (!fini) {

    switch (menu() ) {

    case 0:
      fini = vrai;
      break;

    case 1 : {
      Personne* nouveau = creerPersonne();
      insererEnTeteDeListe (lp, nouveau);
      } break;

    case 2 : {
      Personne* nouveau = creerPersonne();
      insererEnFinDeListe (lp, nouveau);
      } break;

    case 3 : {
      Personne* extrait = (Personne*) extraireEnTeteDeListe (lp);
      if (extrait != NULL) {
        printf ("Élément %s %s extrait en tête de liste",
                extrait->nom, extrait->prenom);
      } else {
        printf ("Liste vide");
      }
      } break;

    case 4 : {
      Personne* extrait = (Personne*) extraireEnFinDeListe (lp);
      if (extrait != NULL) {
        printf ("Élément %s %s extrait en fin de liste",
                extrait->nom, extrait->prenom);
      } else {
        printf ("Liste vide");
      }
      } break;

    case 5 : {
      printf ("Nom de la personne à extraire ? ");
      ch15 nom; scanf ("%s", nom);
```

```
      Personne* cherche = creerPersonne (nom, "?");
      Personne* pp      = (Personne*) chercherUnObjet (lp, cherche);
      booleen extrait   = extraireUnObjet (lp, pp);
      if (extrait) {
        printf ("Élément %s %s extrait de la liste", pp->nom, pp->prenom);
      }
      } break;

    case 6:
      listerListe (lp);
      break;

    case 7 : {
      printf ("Nom de la personne recherchée ? ");
      ch15 nom; scanf ("%s", nom);
      Personne* cherche = creerPersonne (nom, "?");
      Personne* pp      = (Personne*) chercherUnObjet (lp, cherche);
      if (pp != NULL) {
        printf ("%s %s trouvée dans la liste\n", pp->nom, pp->prenom);
      } else {
        printf ("%s inconnue dans la liste\n", nom);
      }
      } break;

    case 8:{
      printf ("1 - Insertion en ordre croissant\n");
      printf ("2 - Insertion en ordre décroissant\n");
      printf ("\nVotre choix ? ");
      int cd; scanf ("%d", &cd);

      FILE* fe = fopen ("noms.dat", "r");
      if (fe==NULL) {
        printf ("Erreur ouverture de noms.dat\n");
      } else {
        lp = creerListe (cd, toStringPersonne, comparerPersonne);
        while ( !feof (fe) ) {
          ch15 nom; ch15 prenom;
          fscanf (fe, "%15s%15s", nom, prenom);
          Personne* nouveau = creerPersonne (nom, prenom);
          insererEnOrdre (lp, nouveau);
        }
        fclose (fe);
        listerListe (lp);
      }
      } break;

    case 9:
      detruireListe (lp);
      break;
    } // switch
  } // while
}
```

Exemple d’exécution, le fichier *noms.dat* contient les informations suivantes :

```
Duval         Albert
Dupont        Julien
Dupond        Michèle
Duvallon      Jacqueline
Duroc         René
```

```
GESTION D'UNE LISTE DE PERSONNES

0 - Fin
1 - Insertion   en tête de liste
2 - Insertion   en fin  de liste
3 - Retrait     en tête de liste
4 - Retrait     en fin  de liste
5 - Retrait     d'un élément à partir de son nom
6 - Parcours    de la liste
7 - Recherche   d'un élément à partir de son nom
8 - Insertion   ordonnée à partir d'un fichier
9 - Destruction de la liste

Votre choix ? 8

1 - Insertion en ordre croissant
2 - Insertion en ordre décroissant

Votre choix ? 1
Dupond Michèle
Dupont Julien
Duroc René
Duval Albert
Duvallon Jacqueline
```

2.4.3 Les polynômes

Il s'agit de mémoriser des polynômes d'une variable réelle et de réaliser des opérations sur ces polynômes. Le nombre de monômes est variable, aussi une allocation dynamique d'espace mémoire s'impose. La gestion en liste facilite l'ajout ou la suppression de monômes pour un polynôme donné.

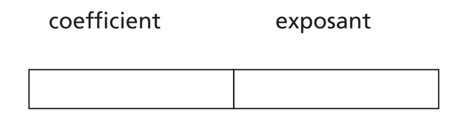


Figure 25 Le type Monome.

Les polynômes suivants :

$A = 3x^5 + 2x^3 + 1$

$B = 6x^5 - 5x^4 - 2x^3 + 8\ x^2$

peuvent être mémorisés comme indiqué sur la Figure 26.

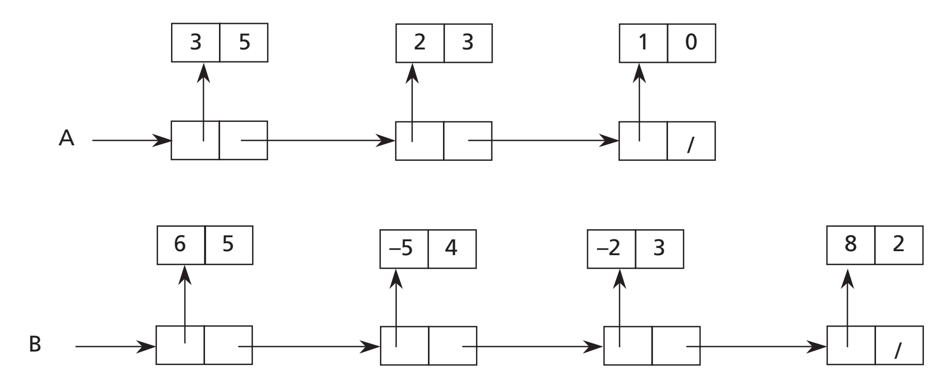


Figure 26 Mémorisation de polynômes sous forme de listes.

On crée un module *polynome* en définissant l'interface *polynome.h* du module (voir Figure 16, page 25). Le type *Monome* est une structure (un objet) contenant un coefficient réel et un exposant entier. Le type *Polynome* correspond au type *Liste* puisqu'on utilise une liste ordonnée pour mémoriser le polynôme. Les fonctions du module polynôme permettent de créer un monôme, d'insérer un monôme dans un polynôme, de lister un polynôme et de calculer la valeur d'un polynôme pour une valeur de x donnée.

2.4.3.a Le fichier d'en-tête des polynômes

Le fichier d'en-tête *polynome.h* décrit l'interface du module des polynômes.

```
// polynome.h

#ifndef POLYNOME_H
#define POLYNOME_H

#include "liste.h"

typedef struct {
  double  coefficient;
  int     exposant;
} Monome;

typedef Liste  Polynome;

Monome*   creerMonome      (double coefficient, int exposant);
Monome*   creerMonome      ();
Polynome* creerPolynome    ();
void      insererEnOrdre   (Polynome* po, Monome* nouveau);
void      listerPolynome   (Polynome* po);
double    valeurPolynome   (Polynome* po, double x);
Monome*   chercherUnMonome (Polynome* po, Monome* nouveau);
booleen   extraireMonome   (Polynome* po, Monome* cherche);
void      detruirePolynome (Polynome* po);
#endif
```

2.4.3.b Le module des polynômes

Le fichier *polynome.cpp* contient le corps des fonctions définies ci-dessus. *listerPolynome()*, *valeurPolynome()* sont des algorithmes de parcours de listes.

```
/* polynome.cpp
   Utilisation du module de gestion des listes */

#include <stdio.h>
#include <stdlib.h>      // exit
#include "polynome.h"

// LES MONOMES

Monome* creerMonome (double coefficient, int exposant) {
  Monome* nouveau      = new Monome();
  nouveau->coefficient = coefficient;
  nouveau->exposant    = exposant;
  return nouveau;
}

// créer un monôme par lecture du coefficient et de l'exposant
Monome* creerMonome () {
  double coefficient;
  int    exposant;
  printf ("Coefficient ? "); scanf ("%lf", &coefficient);
  printf ("Puissance   ? "); scanf ("%d",  &exposant);
  return creerMonome (coefficient, exposant);
}

// écrire un monôme : +3.00 x**5 par exemple
static void ecrireMonome (Monome* monome) {
  printf (" %+.2f x**%d ", monome->coefficient, monome->exposant);
}

// comparer deux monômes m1 et m2
static int comparerMonome (Monome* m1, Monome* m2) {
  if (m1->exposant < m2->exposant) {
    return -1;
  } else if (m1->exposant == m2->exposant) {
    return 0;
  } else {
    return 1;
  }
}

// écrire un objet monôme, pour listerPolynoome()
static void ecrireMonome (Objet* objet) {
  ecrireMonome ( (Monome*) objet);
}

static int comparerMonome (Objet* objet1, Objet* objet2) {
  return comparerMonome ((Monome*)objet1, (Monome*)objet2);
}

Polynome* creerPolynome () {
  return creerListe (DECROISSANT, NULL, comparerMonome);
}

void insererEnOrdre (Polynome* po, Monome* nouveau) {
  // sans (Objet*), le compilateur considère un appel récursif
  insererEnOrdre (po, (Objet*) nouveau);   // du module liste
}
```

```
// puissance nième d'un nombre réel x (n entier >=0)
// voir en 1.2.5 page 10
static double puissance (double x, int n) {
  double resu;
  if (n==0) {
    resu = 1.0;
  } else {
    resu = puissance (x, n/2);
    if (n%2 == 0) {
      resu = resu*resu;     // n pair
    } else {
      resu = resu*resu*x;   // n impair
    }
  }
  return resu;
}

// LES POLYNOMES

// lister le polynôme po
void listerPolynome (Polynome* po) {
  listerListe (po, ecrireMonome);
}

// valeur du polynôme po pour un x donné
double valeurPolynome (Polynome* po, double x) {
  Liste*  li = po;

  double  resu = 0;
  if (listeVide (li) ) {
    printf ("Polynôme nul\n"); exit (1);
  } else {
    ouvrirListe (li);
    while (!finListe (li)) {
      Monome* ptc = (Monome*) objetCourant (li);
      resu  += ptc->coefficient*puissance(x, ptc->exposant);
    }
  }
  return resu;
}

Monome* chercherUnMonome (Polynome* po, Monome* nouveau) {
  return (Monome*) chercherUnObjet (po, nouveau);
}

booleen extraireMonome (Polynome* po, Monome* objet) {
  return extraireUnObjet (po, objet);
}

void detruirePolynome (Polynome* po) {
  detruireListe (po);
}
```

2.4.3.c Le programme principal des polynômes

Le menu et le programme principal suivants permettent de définir un polynôme (une liste ordonnée), d'y insérer des monômes en ordre décroissant des exposants, de lister le polynôme, de calculer la valeur du polynôme pour une valeur donnée, de supprimer un monôme à partir de son exposant ou de détruire la liste du polynôme.

```
/* pppolynome.cpp  programme principal des polynômes
   Utilisation du module de gestion des listes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "polynome.h"

int menu () {
  printf ("\n\nGESTION DE POLYNOMES\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Insertion d'un monôme\n");
  printf ("2 - Écriture du polynôme\n");
  printf ("3 - Valeur du pôlynome pour un x donné\n");
  printf ("4 - Retrait d'un monôme à partir de son exposant\n");
  printf ("5 - Destruction de la liste\n");

  printf ("\nVotre choix ? ");
  int cod; scanf ("%d", &cod);
  printf ("\n");
  return cod;
}

void main () {
  Polynome* po   = creerPolynome();
  booleen   fini = faux;

  while (!fini) {

    switch ( menu() ) {

    case 0:
      fini = vrai;
      break;

    case 1 : {
      Monome* nouveau = creerMonome();
      insererEnOrdre (po, nouveau);
      } break;

    case 2 : {
      listerPolynome (po);
      } break;

    case 3 : {
      printf ("A(x) = "); listerPolynome (po);
      printf ("\nValeur de x ? ");
      double x; scanf ("%lf", &x);
      printf ("A (%.2f) = %.2f\n", x, valeurPolynome (po, x));
      } break;

    case 4 : {
      printf ("Exposant du monôme à extraire ? ");
      int exposant; scanf ("%d", &exposant);
      Monome* cherche = creerMonome (0, exposant);
      Monome* ptc     = chercherUnMonome (po, cherche);
      booleen extrait = extraireMonome (po, ptc);
      if (extrait) {
        printf ("extrait le monôme%.2f x** %d\n",
                  ptc->coefficient, ptc->exposant);
      } else {
        printf ("pas de monôme ayant cet exposant\n");
```

```
      }
      } break;

    case 5 :
      detruirePolynome (po);
      break;
    }  // switch
  }  // while
}
```

L'encadré suivant est un exemple d'exécution de pppolynome.cpp pour la création du polynôme ordonné suivant les puissances décroissantes : A (x) = $3x^5 + 2x^3 + 1$, et pour le calcul de sa valeur pour x=2.

```
GESTION DE POLYNOMES

0 - Fin
1 - Insertion d'un monôme
2 - Écriture du polynôme
3 - Valeur du polynôme pour un x donné
4 - Retrait d'un monôme à partir de son exposant
5 - Destruction de la liste

Votre choix ? 3

A(x) = +3.00 x**5 +2.00 x**3 +1.00 x**0
Valeur de x ? 2
A (2.00) = 113.00
```

Exercice 7 - Polynômes d'une variable réelle (lecture, addition)

Compléter les fonctions du polynôme du § 2.4.3.A, en créant un nouveau fichier d'en-tête `polynome2.h` comme suit :

```
/* polynome2.h */

#include <stdio.h>
#include "polynome.h"

Polynome* lirePolynome (FILE* fe);
Polynome* addPolynome  (Polynome* a, Polynome* b);
Polynome* sousPolynome (Polynome* a, Polynome* b);
```

Écrire les fonctions suivantes (fichier `polynome2.cpp`) :

- *Polynome* lirePolynome (FILE* fe) ;* qui crée un polynôme en lisant les coefficients et les exposants du polynôme dans le fichier fe.
- *Polynome* addPolynome (Polynome* a, Polynome* b) ;* qui fournit le polynôme résultant de l'addition des polynômes a et b.

• *Polynome* sousPolynome (Polynome* a, Polynome* b) ;* qui fournit le polynôme résultant de la soustraction des polynômes a et b.

Écrire un programme de test de *lirePolynome()*, *addPolynome()* et *sousPolynome()*.

2.4.4 Les systèmes experts

2.4.4.a Introduction

Les systèmes experts sont des logiciels fournissant dans un domaine particulier les mêmes conclusions qu'un homme expert en ce domaine : fournir un diagnostic médical à partir d'une liste de symptômes, ou classer des espèces animales ou végétales à partir d'observations par exemple.

Un système expert doit donc :

• enregistrer les faits initiaux (les symptômes d'un malade, les observations sur l'animal en cours d'examen),

• et appliquer des règles générales pour en déduire de nouveaux faits non connus initialement.

Cet exemple est donné pour illustrer l'utilisation des listes, et non pour expliquer les systèmes experts en détail. Les règles mentionnées ci-dessous sont données à titre indicatif, sans prétention quant au domaine de l'expert, mais sur deux exemples de règles pour montrer l'indépendance du logiciel d'inférence (de déduction) de règles avec le domaine traité. Ce logiciel de déduction de nouveaux faits est habituellement appelé moteur d'inférence.

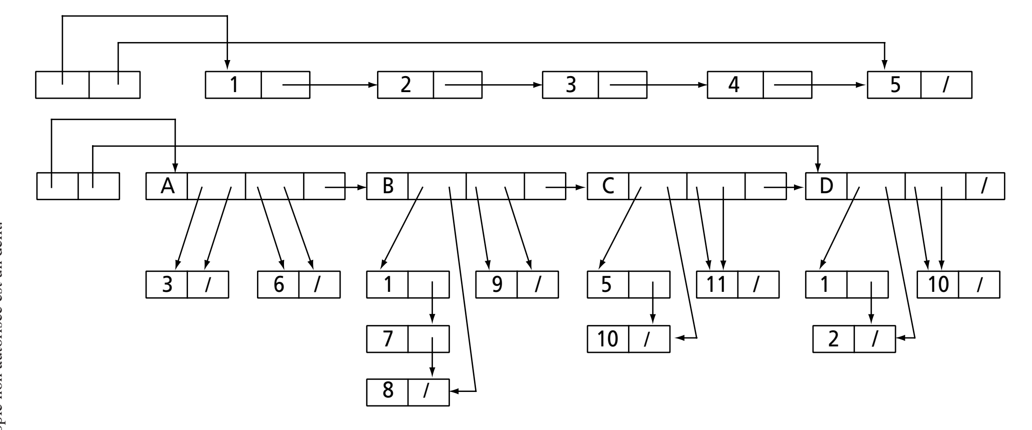


Figure 27 Principe de la mémorisation des faits et des règles dans un système expert. Voir le détail de l'implémentation sur les figures suivantes.

La première liste de la Figure 27 contient les faits initiaux 1, 2, 3, 4, 5. La seconde liste mémorise les règles A, B, C, D. Chaque règle est constituée de son nom, d'une

liste d'hypothèses (1, 7, 8 pour la règle B) et d'une liste de conclusions (9 pour la règle B). Il pourrait y avoir plusieurs conclusions. Les faits hypothèses et conclusions sont indiqués par leur numéro. Les deux tableaux ci-dessous permettent de passer à une application plus concrète et font correspondre un libellé à un numéro. Le premier tableau fait référence à un système expert de diagnostic médical, le second à une classification d'animaux.

1	a de la fièvre
2	a le nez bouché
3	a mal au ventre
4	a des frissons
5	a la gorge rouge
6	a l'appendicite
7	a mal aux oreilles
8	a mal à la gorge
9	a les oreillons
10	a un rhume
11	a la grippe

1	allaite ses petits
2	a des crocs développés
3	vit en compagnie de l'homme
4	grimpe aux arbres
5	a des griffes acérées
6	est domestiqué
7	est couvert de poils
8	a quatre pattes
9	est un mammifère
10	est un carnivore
11	est un chat

La règle B pourrait se formuler comme suit :

```
B – si
  l'animal allaite ses petits
  et l'animal est couvert de poils
  et l'animal a quatre pattes
alors
  l'animal est un mammifère
```

À partir des faits initiaux 1, 2, 3, 4, 5, et en appliquant les règles, on peut ajouter pour la règle A dont l'hypothèse 3 est donnée comme fait initial, le fait conclusion 6. La règle B ne s'applique pas, seule l'hypothèse 1 est vérifiée. La règle C ne s'applique pas, seule l'hypothèse 5 est vérifiée. La règle D s'applique, car les hypothèses 1 et 2 sont données comme faits initiaux. Le fait 10 est ajouté à la liste de faits. Il faut refaire un parcours des règles et voir si, suite à l'adjonction de nouveaux faits, de nouvelles règles ne sont pas vérifiées.

C'est le cas de la règle C qui est vérifiée au deuxième passage car le fait 10 a été ajouté par la règle D. Le fait 11 est donc ajouté. Un nouveau parcours des règles n'entraîne aucun ajout. Cette façon de procéder s'appelle le chaînage avant. Si les règles sont nombreuses, on risque de déduire de nombreux faits nouveaux, difficilement exploitables.

Une autre façon de procéder consiste à demander si le système ne peut pas démontrer un fait. Sur la Figure 27, peut-on démontrer le fait 11 ? Si le fait 11 n'est pas donné comme fait initial, il faut trouver une règle qui a 11 pour conclusion, et

essayer de démontrer ses hypothèses. Pour démontrer 11, il faut démontrer 5 et 10 (règle C). 5 est un fait initial, reste à démontrer 10. Pour démontrer 10 (règle D), il faut démontrer 1 et 2 qui sont des faits initiaux. Donc 11 est vrai (démontré). Cette méthode s'appelle le chaînage arrière (voir Figure 28).

Démontrer le fait 11 sur les deux exemples donnés consiste à démontrer que l'animal *est un chat*, ou que le patient *à la grippe*.

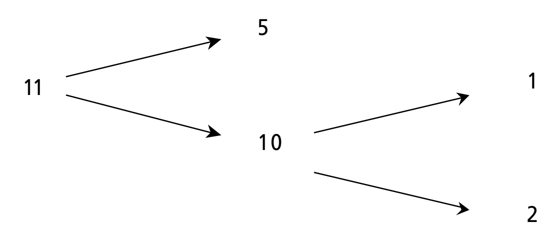


Figure 28 Chaînage arrière dans un système expert.

2.4.4.b Listes de faits et liste de règles

Les structures de données de la Figure 27 sont décrites ci-dessous, ainsi que les fonctions de gestion des faits initiaux et des règles. Le module de gestion de liste est utilisé sans modification pour les listes de faits (faits initiaux, hypothèses, conclusions) et pour la liste des règles. Le champ *marque* pour une règle est vrai si la règle s'est déjà exécutée ; il est alors inutile de la tester lors des passages suivants. *creerRegle()* initialise une règle à l'aide de son nom, la règle n'ayant aucune hypothèse ni aucune conclusion. *ajouterFait()* ajoute un fait à une liste de faits comme par exemple la liste des faits initiaux, la liste des faits hypothèses ou la liste des faits conclusions. *listerFait()* et *listerLesRegles()* sont des parcours de listes.

```
// systexpert.cpp   système expert

#include <stdio.h>
#include <string.h>
#include "liste.h"

typedef char ch3 [3];
char* message (int n);  // fournit le libellé du fait n

typedef     Liste  ListeFaits;
typedef     Liste  ListeRegles;
void        ajouterFait     (ListeFaits* listF, int n);
void        listerFaits     (ListeFaits* listF);
ListeFaits* creerListeFaits ();
```

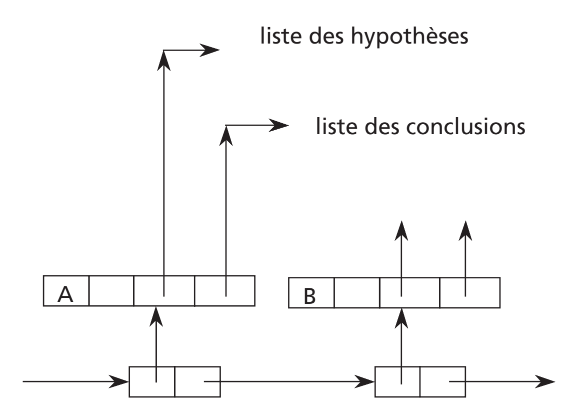


Figure 29 Détails de l'implémentation d'une règle.

```
// LES REGLES

typedef struct {
  ch3          nom;
  booleen      marque;
  ListeFaits*  hypotheses;
  ListeFaits*  conclusions;
} Regle;


// constructeur d'une règle à partir de son nom;
// les listes hypothèses et conclusions sont vides
Regle* creerRegle (ch3 nom) {
  Regle* regle = new Regle();
  strcpy (regle->nom, nom);
  regle->hypotheses  = creerListeFaits();
  regle->conclusions = creerListeFaits();
  regle->marque = faux;
  return regle;
}

// ajouter le fait n aux hypothèses de la règle "regle"
void ajouterHypothese (Regle* regle, int n) {
  ajouterFait (regle->hypotheses, n);
}

// ajouter le fait n aux conclusions de la règle "regle"
void ajouterConclusion (Regle* regle, int n) {
  ajouterFait (regle->conclusions, n);
}

// lister la règle "regle"
void listerUneRegle (Regle* regle) {
  printf ("\nRègle : %s\n", regle->nom);
  printf ("  hypothèses\n");
  listerFaits (regle->hypotheses);
  printf ("  conclusions\n");
  listerFaits (regle->conclusions);
}
```

```
// lister toutes les règles
void listerLesRegles (ListeRegles* lr) {
  ouvrirListe (lr);
  while (!finListe(lr) ) {
    Regle* ptc = (Regle*) objetCourant (lr);
    listerUneRegle (ptc);
  }
  printf ("\n");
}
```

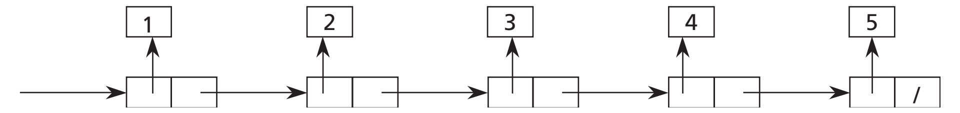


Figure 30 Détails de l'implémentation d'une liste de faits.

```
// LES FAITS

typedef struct {
  int  numero;
} Fait;


// constructeur de Fait
Fait* creerFait (int n) {
  Fait* nouveau = new Fait();
  nouveau->numero = n;
  return nouveau;
}



// LES LISTES DE FAITS

ListeFaits* creerListeFaits() {
  return creerListe();
}

// ajouter le fait n à la liste de faits listF
void ajouterFait (ListeFaits* listF, int n) {
  Fait* nouveau = creerFait (n);
  insererEnFinDeListe (listF, nouveau);
}


// lister les faits de la liste listF
void listerFaits (ListeFaits* listF) {
  ouvrirListe (listF);
  while (!finListe(listF) ) {
    Fait* ptc = (Fait*) objetCourant (listF);
    printf ("    %s\n", message (ptc->numero));
  }
}
```

Remarque : si on veut mémoriser le numéro de l'entier dans le champ reference, plutôt que le pointeur vers l'entier (voir Figure 30), il suffit de remplacer :

```
Fait* creerFait (int n) {
  return (Fait*) n;          // n doit être considéré comme un pointeur
}
```

et de remplacer :

```
ptc->numero par (int)ptc      ptc doit être considéré comme un entier
```

Exercice 8 - Systèmes experts : les algorithmes de déduction

- Écrire la fonction : *booleen existe (ListeFaits* listF, int num) ;* qui indique si le fait num existe dans la liste listF.
- Écrire la fonction : *int appliquer (Regle* regle, ListeFaits* listF) ;* qui vérifie si la règle pointée par regle s'applique, et ajoute les conclusions de cette règle à la liste de faits listF si les hypothèses de la règle sont vérifiées.
- Écrire la fonction : *void chainageAvant (ListeRegles* listR, ListeFaits* listF) ;* qui à partir de la liste de faits listF et de la liste des règles listR, ajoute à listF les conclusions des règles vérifiées.
- Écrire la fonction récursive : *booleen demontrerFait (ListeRegles* listR, ListeFaits* listF, int num, int nb) ;* qui en utilisant la liste de faits listF et la liste des règles listR, démontre le fait num ; nb est utilisé pour faire une indentation (au fil des appels récursifs) comme sur le schéma de la Figure 28.
- Écrire le programme principal qui crée les structures de données de la Figure 27, liste les faits et les règles, effectue le chaînage avant et le chaînage arrière.

2.4.5 Les piles

Une pile est une structure de données telle que :

- l'ajout d'un élément se fait au sommet de la pile,
- la suppression d'un élément se fait également au sommet de la pile.

La structure de données est appelée LIFO : "last in, first out" soit "dernier entré, premier sorti".

2.4.5.a Allocation dynamique (utilisation de listes)

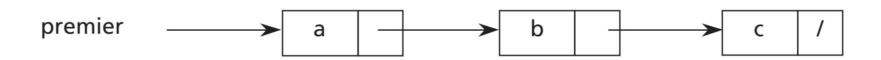


Figure 31 Principe d'une pile gérée à l'aide d'une liste.

À l'aide d'une liste, les opérations sur une pile peuvent être réalisées comme suit :

initialiserPile	initialiser une liste vide
pileVide	vrai si la liste est vide
empiler	ajouter un élément en tête de la liste
dépiler	enlever un élément en tête de la liste

Le module de gestion de piles peut facilement se réaliser avec le module de gestion des listes présenté précédemment en utilisant *insererEnTeteDeListe()* et *extraireEnTeteDeListe()*. Un élément de la liste référence un objet spécifique de l'application. Les déclarations et les opérations sur la pile sont données ci-dessous.

2.4.5.b Le fichier d'en-tête des piles (utilisation de listes)

pile.h contient la déclaration du type pile et les prototypes des fonctions de gestion de la pile.

```
// pile.h  pile en allocation dynamique avec des listes

#ifndef PILE_H
#define PILE_H

#include "liste.h"

typedef  Liste Pile;

Pile*    creerPile    ();
booleen  pileVide     (Pile* p);
void     empiler      (Pile* p, Objet* objet);
Objet*   depiler      (Pile* p);
void     listerPile   (Pile* p, void (*f) (Objet*));
void     detruirePile (Pile* p);

#endif
```

2.4.5.c Le module des piles (utilisation de listes)

pile.cpp contient le corps des fonctions dont le prototype est défini dans *pile.h*.

```
/* pile.cpp  pile gérée à l'aide d'une liste */

#include <stdio.h>
#include <stdlib.h>
#include "pile.h"

// créer et initialiser une pile
Pile* creerPile () {
  return creerListe ();
}

// vrai si la pile est vide, faux sinon
booleen pileVide (Pile* p) {
  return listeVide (p);
}

// empiler objet dans la pile p
void empiler (Pile* p, Objet* objet) {
  insererEnTeteDeListe (p, objet);
}

// fournir l'adresse de l'objet en sommet de pile,
// ou NULL si la pile est vide
Objet* depiler (Pile* p) {
  if (pileVide (p)) {
```

```
    return NULL;
  } else {
    return extraireEnTeteDeListe (p);
  }
}

// Lister la pile du sommet vers la base
void listerPile (Pile* p, void (*f) (Objet*)) {
  listerListe (p, f);
}

void detruirePile (Pile* p) {
  detruireListe (p);
}
```

2.4.5.d Déclaration des types Entier et Reel pour le test de la pile

Les types Entier et Reel sont utilisés à plusieurs reprises dans la suite de ce livre. Les déclarations et le corps des fonctions traitant de ces types sont insérés dans les fichiers *mdtypes.h* et *mdtypes.cpp* (voir § 2.4.1, page 51), après les déclarations du type Personne.

Dans *mdtypes.h* :

```
// **** une structure contenant un entier
typedef struct {
  int valeur;
} Entier;

Entier*   creerEntier       (int valeur);
Entier*   entier            (int valeur);
void      ecrireEntier      (Objet* objet);
char*     toStringEntier    (Objet* objet);
int       comparerEntier    (Objet* objet1, Objet* objet2);
int       comparerEntierCar (Objet* objet1, Objet* objet2);


// **** une structure contenant un réel double
typedef struct {
  double valeur;
} Reel;

Reel*     creerReel        (double valeur);
void      ecrireReel       (Objet* objet);
int       comparerReel     (Objet* objet1, Objet* objet2);
```

Dans *mdtypes.cpp* :

```
// **** une structure contenant un entier

Entier* creerEntier (int valeur) {
  Entier* entier = new Entier();
  entier->valeur = valeur;
  return entier;
}

void ecrireEntier (Objet* objet) {
  Entier* entier = (Entier*) objet;
  printf ("%d\n", entier->valeur);
}
```

```
// constructeur de Entier
Entier* entier (int valeur) {
  return creerEntier (valeur);
}

char* toStringEntier (Objet* objet) {
  char* nombre = (char*) malloc (50);
  sprintf (nombre, "%d", ((Entier*)objet)->valeur);
  return nombre;
}


// comparer deux entiers
// fournit <0 si e1 < e2; 0 si e1=e2; >0 sinon
int comparerEntier (Objet* objet1, Objet* objet2) {
  Entier* e1 = (Entier*) objet1;
  Entier* e2 = (Entier*) objet2;
  if (e1->valeur < e2->valeur) {
    return -1;
  } else if (e1->valeur == e2->valeur) {
    return 0;
  } else {
    return 1;
  }
}

// comparer des chaînes de caractères correspondant à des entiers
// 9 < 100 (mais pas en ascii)
int comparerEntierCar (Objet* objet1, Objet* objet2) {
  long a = atoi ((char*) objet1);
  long b = atoi ((char*) objet2);
  if (a==b) {
    return 0;
  } else if (a<b) {
    return -1;
  } else {
    return 1;
  }
}


// **** une structure contenant un réel double

Reel* creerReel (double valeur) {
  Reel* reel = new Reel();
  reel->valeur = valeur;
  return reel;
}

void ecrireReel (Objet* objet) {
  Reel* reel = (Reel*) objet;
  printf ("%.2f\n", reel->valeur);
}
```

2.4.5.e Utilisation du module de gestion de piles

Le programme suivant définit un menu et un programme principal permettant d'initialiser une pile, de tester si la pile est vide, d'ajouter ou de retirer des éléments

en sommet de pile et de lister pour vérification, le contenu de la pile. Le module pile.h peut être utilisé pour n'importe quel objet à empiler (entier, réel, personne, etc.). Pour cet exemple, repris dans l'exercice suivant, la variable de compilation PILETABLEAU n'est pas définie.

```
// pppile.cpp  programme principal des piles (avec listes ou tableau)

#include <stdio.h>

#ifdef PILETABLEAU
#include "piletableau.h"
#else
#include "pile.h"
#endif

#include "mdtypes.h"

int menu () {
  printf ("\n\nGESTION D'UNE PILE D'ENTIERS\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Initialisation de la pile\n");
  printf ("2 - La pile est-elle vide\n");
  printf ("3 - Insertion   dans la pile\n");
  printf ("4 - Retrait     de   la pile\n");
  printf ("5 - Listage     de   la pile\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  printf ("\n");
  return cod;
}

void main () {
  #ifdef PILETABLEAU
  #define LGMAX 7
  Pile*   pile1 = creerPile(LGMAX);
  #else
  Pile*   pile1 = creerPile();
  #endif
  booleen fini = faux;

  while (!fini) {

    switch (menu() ) {

    case 0:
      fini = vrai;
      break;

    case 1:
      detruirePile (pile1);
      #ifdef PILETABLEAU
      pile1 = creerPile(LGMAX);
      printf ("pile = un tableau de %d places\n", LGMAX);
      #else
      pile1 = creerPile();
      #endif
      break;
```

```
    case 2:
      if (pileVide (pile1) ) {
        printf ("Pile vide\n");
      } else {
        printf ("Pile non vide\n");
      }
      break;

    case 3 : {
      int valeur;
      printf ("Valeur à empiler ? ");
      scanf ("%d", &valeur);
      empiler  (pile1, creerEntier(valeur));
      } break;

    case 4 : {
      Entier* v;
      if ((v = (Entier*) depiler (pile1)) != NULL) {
        ecrireEntier (v);
      } else {
        printf ("Pile vide\n");
      }
      } break;

    case 5:
      listerPile (pile1, ecrireEntier);
      break;
    }  // switch
  }

  detruirePile (pile1);


  printf ("\n\nGESTION D'UNE PILE DE PERSONNES\n");
  #if PILETABLEAU
  printf ("avec un tableau de %d places\n", LGMAX);
  Pile*   pile2 = creerPile(LGMAX);
  #else
  Pile*   pile2 = creerPile();
  #endif

  empiler (pile2, creerPersonne("Dupont",   "Jacques"));
  empiler (pile2, creerPersonne("Dufour",   "Jacques"));
  empiler (pile2, creerPersonne("Dupré",    "Jacques"));
  empiler (pile2, creerPersonne("Dumoulin", "Jacques"));
  printf ("Valeurs dans la pile : du sommet vers la base\n");
  listerPile (pile2, ecrirePersonne);

  printf ("\nValeur dépilée : ");
  Personne* p = (Personne*) depiler (pile2);
  if (p!=NULL) ecrirePersonne (p);


  printf ("\n\nGESTION D'UNE PILE DE REELS\n");
  #if PILETABLEAU
  printf ("avec un tableau de %d places\n", LGMAX);
  Pile*   pile3 = creerPile(7);
  #else
  Pile*   pile3 = creerPile();
  #endif
  empiler (pile3, creerReel (2.5));
  empiler (pile3, creerReel (3.5));
```

```
    empiler (pile3, creerReel (5.5));
    printf ("Valeurs dans la pile : du sommet vers la base\n");
    listerPile (pile3, ecrireReel);

    printf ("\nvaleur dépilée : ");
    Reel* r = (Reel*) depiler (pile3);
    if (r!=NULL) ecrireReel (r);
}
```

2.4.5.f Allocation contiguë (utilisation d'un tableau)

Les piles peuvent être également gérées à l'aide d'un tableau alloué sur des cases contiguës et de taille a priori connue et donc limitée (à 7 sur la Figure 32). Les éléments sont consécutifs en mémoire. Chaque élément du tableau contient un pointeur sur un objet de la pile. La pile peut être une pile d'entiers, de réels, de personnes comme précédemment.

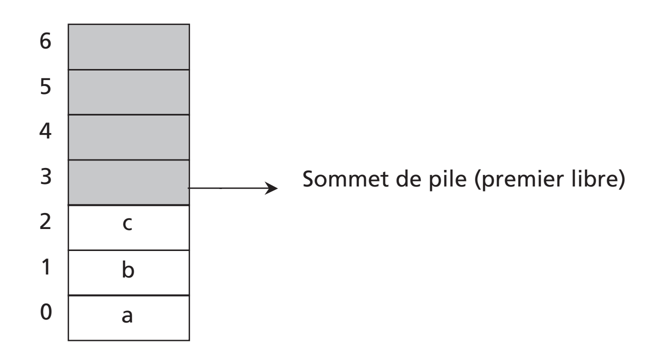


Figure 32 Principe d'une pile gérée à l'aide d'un tableau.

Exercice 9 - Le type pile (allocation contiguë)

Reprendre la déclaration de *pile.h* du § 2.4.5.b, page 67 et le module *pile.cpp* correspondant de façon à gérer la pile à l'aide d'un tableau. Tester le programme utilisateur *pppile.cpp* utilisé ci-dessus pour l'allocation dynamique en liste qui doit rester inchangé sauf pour *creerPile()* qui contient un paramètre indiquant la taille de la pile dans le cas de l'allocation contiguë. Le type pile est un TAD (Type Abstrait de Données) ; son implémentation ne doit pas affecter les programmes utilisateurs.

2.4.6 Les files d'attente (gérée à l'aide d'une liste)

Une file d'attente est une structure de données telle que :

- l'ajout d'un élément se fait en fin de file d'attente,
- la suppression d'un élément se fait en début de file d'attente.

La structure de données est appelée FIFO : « first in, first out » soit « premier entré, premier sorti ».

2.4.6.a Allocation dynamique (utilisation de listes)

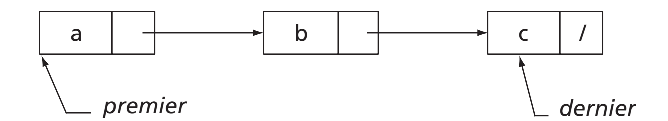


Figure 33 Principe d'une file d'attente gérée à l'aide d'une liste.

À l'aide d'une liste, les opérations sur une file d'attente peuvent être réalisées comme suit :

créerFile	créer et initialiser une liste vide
fileVide	vrai si la liste est vide
enfiler	ajouter un élément en fin de la liste
défiler	enlever un élément en tête de la liste

Le module de gestion des files d'attente peut facilement se réaliser avec le module de gestion des listes en utilisant les fonctions *insererEnFinDeListe()* et *extraireEnTeteDeListe()*.

Exercice 10 - Files d'attente en allocation dynamique

Soit le fichier d'en-tête suivant :

```
/* file.h    file d'attente en allocation dynamique */

#ifndef FILE_H
#define FILE_H

#include "liste.h"

typedef Liste File;

File*    creerFile    ();
booleen  fileVide     (File* file);
void     enFiler      (File* file, Objet* objet);
Objet*   deFiler      (File* file);
void     listerFile   (File* file, void (*f) (Objet*));
void     detruireFile (File* file);

#endif
```

En vous inspirant des programmes précédents pour le type *pile* en allocation dynamique et du fichier d'en-tête *file.h* ci-dessus, écrire un module de gestion de files d'attente *file.cpp* et un programme principal de test *ppfile.cpp*.

2.4.6.b Allocation contiguë d'une file d'attente (utilisation d'un tableau)

Les files peuvent être également gérées à l'aide d'un tableau de taille a priori connue, et donc limitée (à 7 sur la Figure 34). Les éléments sont consécutifs en mémoire. *premier* repère l'élément qui précède le premier élément. *dernier* repère le

dernier élément. La file est vide si *premier* est égal à *dernier*. La file est dite pleine si *dernier* précède immédiatement *premier*. Il reste en fait une place inutilisée.

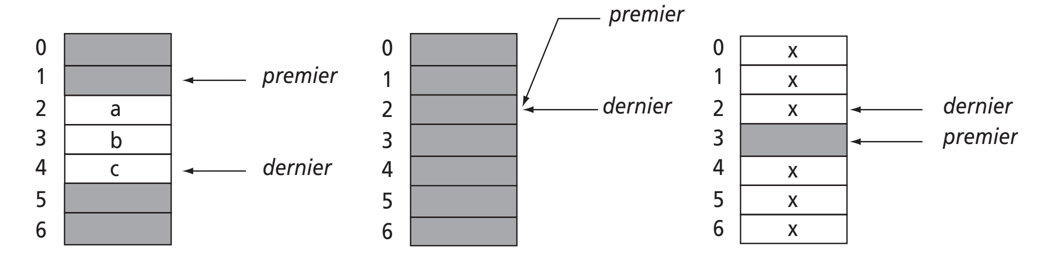


Figure 34 Principe d'une file d'attente gérée à l'aide d'un tableau.

Exercice 11 - Files d'attente en allocation contiguë

Écrire le module *filetableau.cpp* correspondant à la description suivante du fichier d'en-tête *filetableau.h* et réalisant la gestion de files d'attente mémorisées sous forme de tableaux. Le fichier de test *ppfile.cpp* doit être le même que pour l'exercice précédent sauf pour *creerFile()* qui contient un paramètre indiquant la taille du tableau. Le changement de structures de données pour mémoriser les files d'attente ne doit pas perturber les programmes utilisateurs qui considèrent la file comme un type abstrait de données (TAD).

```
/* filetableau.h */

#ifndef FILE_H
#define FILE_H

typedef int     booleen;
#define faux    0
#define vrai    1

typedef void Objet;

typedef struct {
  int     max;        // nombre max d'éléments dans la file
  int     premier;    // élément précédant le premier
  int     dernier;    // dernier occupé
  Objet** element;    // tableau alloué dynamiquement dans creerFile()
} File;

File*    creerFile    (int max);
booleen  fileVide     (File* file);
void     enFiler      (File* file, Objet* objet);
Objet*   deFiler      (File* file);
void     listerFile   (File* file, void (*f) (Objet*));
void     detruireFile (File* file);

#endif
```

2.5 AVANTAGES ET INCONVÉNIENTS DES LISTES

Une liste est une structure de données qui permet la résolution, de façon simple et naturelle en utilisant le module de gestion de listes, de problèmes où l'information est changeante ou de taille difficilement évaluable. Les données sont dispersées en mémoire et reliées seulement par des pointeurs.

La structure de liste présente les avantages suivants :

- l'ajout ou le retrait d'un élément de la liste est facile (simple modification de pointeurs),
- les éléments n'ont pas besoin d'être sur des cases contiguës (que ce soit en mémoire centrale ou sur disque),
- en cas d'allocation dynamique, on n'a pas besoin d'indiquer a priori, le nombre maximum d'éléments (pour la réservation de place). On demande de l'espace au fur et à mesure des besoins.

Cette structure a également quelques inconvénients :

- il y a perte de place pour ranger les pointeurs qui s'ajoutent aux informations caractéristiques de l'application. Avec l'augmentation sur ordinateur des tailles des mémoires centrales ou secondaires, cet inconvénient devient mineur.
- l'accès à un élément ne peut se faire qu'en examinant séquentiellement ceux qui précèdent. Ceci est beaucoup plus gênant. Si la liste est longue et souvent consultée, cette structure devient inefficace, et il faut prévoir d'autres structures privilégiant l'accès rapide à l'information.

2.6 LE TYPE ABSTRAIT DE DONNÉES (TAD) LISTE

L'utilisateur du module de gestion de listes doit utiliser uniquement les fonctions de l'interface du module. Il doit faire abstraction des structures gérées par le module de gestion de listes qui devient un type abstrait de données (TAD). L'utilisateur ne doit jamais accéder directement aux éléments de la tête de liste (premier, dernier, courant), ni au champ suivant d'un élément de liste. Ainsi, pour le type *pile* ou le type *file* des exemples précédents, ce concept de type abstrait de données fait que le programme de test est inchangé (sauf pour la fonction d'initialisation) quand on passe d'une allocation dynamique sous forme de listes à une allocation contiguë sous forme de tableaux. Les programmes utilisateurs ne sont pas de cette façon affectés par un changement des structures de données du module. Cette notion est reprise de manière plus systématique en programmation (orientée) objet par encapsulation des données (voir § 1.4.1 page 24).

2.7 LES LISTES CIRCULAIRES

Une liste circulaire est une liste telle que le dernier élément de la liste contient un pointeur sur le premier (et non une marque de fin comme pour une liste simple). On

peut ainsi parcourir toute la liste à partir de n'importe quel élément. Il faut pouvoir identifier la tête de liste (soit par un pointeur, soit par une marque spéciale dans l'élément de tête).

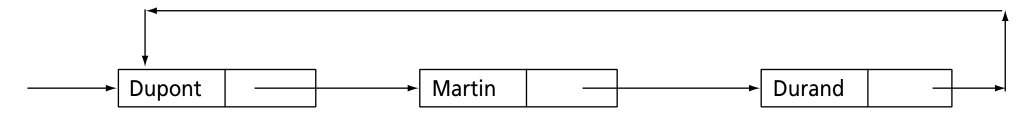


Figure 35 Une liste circulaire de personnes.

2.7.1 Le fichier d'en-tête des listes circulaires

Il est avantageux de remplacer le pointeur de tête par un pointeur sur le dernier élément, ce qui donne facilement accès au dernier, et au premier élément qui est le suivant du dernier. Un élément de liste est défini comme pour les listes simples par un pointeur sur l'objet de l'application et un pointeur sur le suivant. *initListeC()* initialise une liste circulaire. *insererEnTeteDeListeC()* et *insererEnFinDeListeC()* réalisent respectivement l'insertion en tête et en fin de liste circulaire. La fonction *suivant()* fournit un pointeur sur l'élément suivant celui en paramètre. Pour le parcours, il n'y a plus de fin de liste puisque la liste est circulaire.

```
/* listec.h
   La liste circulaire est repérée par un pointeur
   sur le dernier élément
*/
#ifndef LISTEC_H
#define LISTEC_H

typedef void Objet;

typedef struct element {
  Objet*          reference;
  struct element* suivant;
} Element;

typedef struct {
  Element* dernier;
} ListeC;

void     initListeC            (ListeC* lc);
ListeC*  creerListeC           ();

void     insererEnTeteDeListeC (ListeC* lc, Objet* objet);
void     insererEnFinDeListeC  (ListeC* lc, Objet* objet);

// parcours
Element* premier               (ListeC* lc);
Element* dernier               (ListeC* lc);
Element* suivant               (Element* elt);
void     parcoursListeC        (Element* depart, void (*f) (Objet*));

#endif
```

2.7.2 Insertion en tête de liste circulaire

La fonction *insererEnTeteDeListeC (ListeC* lc, Objet* objet) ;* insère l'objet objet en tête de la liste circulaire pointée par lc. L'objet a été alloué avant cet appel et contient les informations spécifiques de l'application. Si la liste est vide, l'élément nouveau pointe sur lui-même, d'où l'instruction nouveau->suivant = nouveau. Le pointeur lc->dernier->suivant repère le premier élément de la liste : c'est le suivant du dernier.

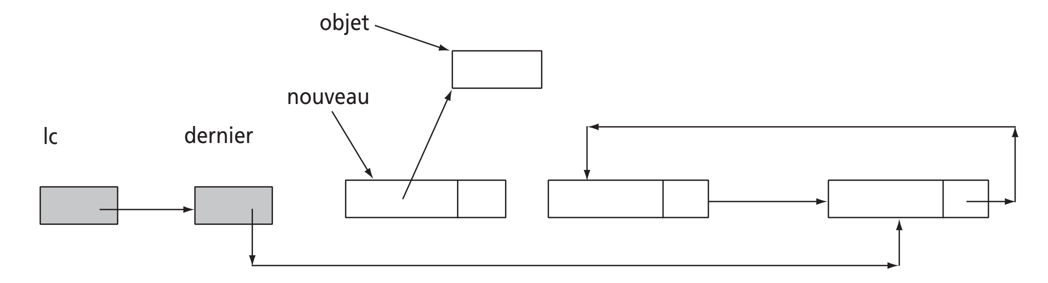


Figure 36 Insertion en tête de liste circulaire.

```
/* listec.cpp
   La liste circulaire est repérée par un pointeur
   sur le dernier élément */

#include <stdio.h>       // NULL
#include "listec.h"

// initialiser une liste circulaire
void initListeC (ListeC* lc) {
  lc->dernier = NULL;
}

ListeC* creerListeC () {
  ListeC* lc = new ListeC();
  initListeC (lc);
  return lc;
}

// créer un élément de liste
static Element* creerElement () {
  return new Element();
}

// ajouter "objet" en début de la liste circulaire lc
void insererEnTeteDeListeC (ListeC* lc, Objet* objet) {
  Element* nouveau   = creerElement();
  nouveau->reference = objet;
  if (lc->dernier == NULL) {      // lc est vide
    nouveau->suivant = nouveau;
    lc->dernier      = nouveau;
  } else {
    nouveau->suivant    = lc->dernier->suivant;
    lc->dernier->suivant = nouveau;
  }
}
```

2.7.3 Insertion en fin de liste circulaire

La fonction : *void insererEnFinDeListeC (ListeC* lc, Objet* objet)* insère objet en fin de la liste circulaire pointée par lc (voir Figure 37).

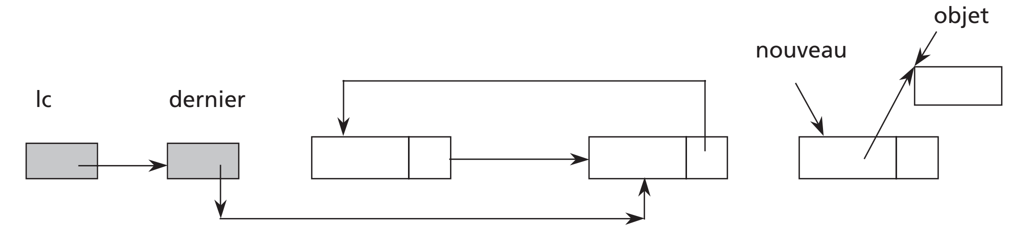


Figure 37 Insertion en fin de liste circulaire.

```
// ajouter "objet" en fin de la liste circulaire lc
void insererEnFinDeListeC (ListeC* lc, Objet* objet) {
  Element* nouveau   = creerElement();
  nouveau->reference = objet;
  if (lc->dernier == NULL) {     // liste vide
    nouveau->suivant = nouveau;
    lc->dernier      = nouveau;
  } else {
    nouveau->suivant     = lc->dernier->suivant;
    lc->dernier->suivant = nouveau;
    lc->dernier          = nouveau;
  }
}
```

2.7.4 Parcours de listes circulaires

```
// le premier est le suivant du dernier
Element* premier (ListeC* lc) {
  return lc->dernier->suivant;
}

Element* dernier (ListeC* lc) {
  return lc->dernier;
}

// fournir un pointeur sur le suivant de elt
Element* suivant (Element* elt) {
  if (elt == NULL) {
    return NULL;
  } else {
    return elt->suivant;
  }
}

// parcours de la liste circulaire en partant de l'élément depart
void parcoursListeC (Element* depart, void (*f) (Objet*)) {
  if (depart == NULL) {
    printf ("Liste circulaire vide\n");
  } else {
    f (depart->reference);
    Element* ptc = suivant (depart);
```

```
    while (ptc != depart) {
      f (ptc->reference);
      ptc = suivant (ptc);
    }
    printf ("\n");
  }
}
```

2.7.5 Le module des listes circulaires

Le module est donné de manière succincte de façon à illustrer les fonctions élémentaires sur les listes circulaires. On pourrait y ajouter de nombreuses fonctions.

```
/* listec.cpp
   La liste circulaire est repérée par un pointeur
   sur le dernier élément */

#include <stdio.h>       // NULL
#include "listec.h"

plus le corps des fonctions définies ci-dessus pour les listes circulaires
```

2.7.6 Utilisation du module des listes circulaires

On peut parcourir toute la liste à partir de n'importe quel élément. On arrête quand on retombe sur l'élément de départ. La liste traitée est une liste de personnes comme au § 2.4.1, page 51. La fonction *ajouterPersonne()* crée une personne à partir de son nom et prénom et l'insère en fin de liste circulaire. La fonction *parcoursListeC()* permet de lister tous les éléments de la liste en partant de n'importe quel élément.

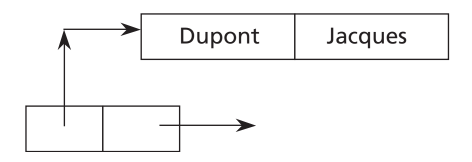


Figure 38 Liste circulaire de Personne.

```
/* pplistec.cpp   programme principal listec */

#include <stdio.h>
#include "listec.h"
#include "mdtypes.h"

// ajouter une personne en fin de liste circulaire
void ajouterPersonne (ListeC* lc, char* nom, char* prenom) {
  Personne* nouveau = creerPersonne (nom, prenom);
  insererEnFinDeListeC (lc, nouveau);
}

void main () {
  ListeC* lc = creerListeC();

  ajouterPersonne (lc, "Dupont", "Jacques");
  ajouterPersonne (lc, "Duroc",  "Albin");
  ajouterPersonne (lc, "Dufour", "Michèle");
```

```
  // parcours à partir du premier de la liste
  printf ("En partant du premier\n");
  parcoursListeC (premier (lc), ecrirePersonne);
  printf ("En partant du dernier\n");
  parcoursListeC (dernier (lc), ecrirePersonne);
}
```

2.8 LES LISTES SYMÉTRIQUES

Une liste symétrique est une liste telle que chaque élément pointe sur l'élément suivant et sur l'élément précédent.

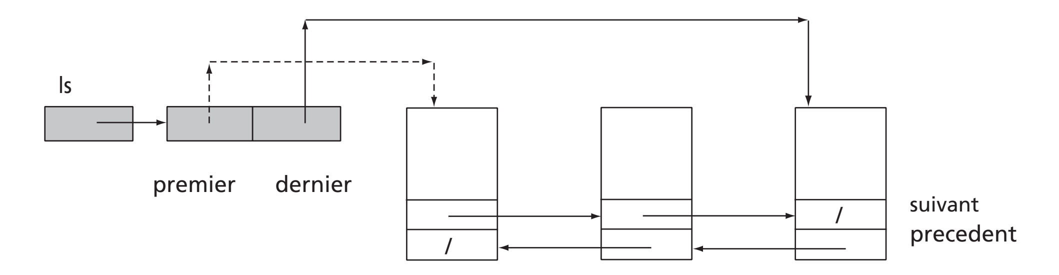


Figure 39 Une liste symétrique.

On peut aussi définir des listes symétriques circulaires. On peut regrouper les informations sur la liste (premier, dernier) dans une tête de liste de type ListeS. L'intérêt majeur des listes symétriques réside dans le fait qu'il est facile d'extraire un élément à partir d'un pointeur sur l'élément à extraire. Il n'y a pas besoin de parcourir la liste pour retrouver le précédent. La liste symétrique peut se trouver en mémoire centrale ou en mémoire secondaire (fichier en accès direct).

2.8.1 Le fichier d'en-tête des listes symétriques

Chaque élément de la liste contient un pointeur sur l'objet de la liste, un pointeur sur l'élément suivant comme pour les listes simples, et un pointeur sur l'élément precedent. Le type *ListeS* est une tête de liste contenant un pointeur sur le premier et un pointeur sur le dernier élément de la liste symétrique. Les fonctions de gestion de la liste permettent de créer et d'initialiser une liste, de savoir si la liste est vide ou pas, de se positionner sur le premier ou sur le dernier élément, d'insérer ou d'extraire des éléments, et de parcourir la liste en demandant le suivant d'un élément ou son précédent. On peut parcourir la liste dans les deux sens.

```
/* listesym.h  Gestion des listes symétriques */

#ifndef LISTESYM_H
#define LISTESYM_H

#include <stdio.h>   // NULL
```

```
typedef int  booleen;
#define faux 0
#define vrai 1
typedef void Objet;

typedef struct element* PElement;
typedef struct element {
  Objet*   reference;
  PElement suivant;
  PElement precedent;
} Element;

typedef struct {
  Element* premier;
  Element* dernier;
  char*    (*toString) (Objet*);
  int      (*comparer) (Objet*, Objet*);
} ListeS;

void     initListeSym           (ListeS* ls, char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*));
ListeS*  creerListeSym          (char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*));
ListeS*  creerListeSym          ();
booleen  listeVide              (ListeS*  ls);

void     insererEnFinDeListeSym (ListeS*  ls, Objet* objet);

Element* premier                (ListeS*  ls);
Element* dernier                (ListeS*  ls);
Element* suivant                (Element* elt);
Element* precedent              (Element* elt);

void     parcoursListeSym       (ListeS* ls, void (*f) (Objet*));
void     parcoursListeSymI      (ListeS* ls, void (*f) (Objet*));

Objet*   chercherObjet          (ListeS* ls, Objet* objet);
void     extraireListeSym       (ListeS* ls, Objet* objet);

#endif
```

2.8.2 Le module des listes symétriques

```
/* listesym.cpp  module des listes symétriques */

#include <stdio.h>
#include <string.h>      // strcmp
#include "listesym.h"

// comparer deux chaînes de caractères
// fournit <0 si ch1 < ch2; 0 si ch1=ch2; >0 sinon
static int comparerCar (Objet* objet1, Objet* objet2) {
  return strcmp ((char*)objet1, (char*)objet2);
}

static char* toChar (Objet* objet) {
  return (char*) objet;
}
```

```
static Element* creerElement () {
  return new Element();
}

// initialiser une liste symétrique
void initListeSym (ListeS* ls, char* (*toString) (Objet*),
                                      int (*comparer) (Objet*, Objet*)) {
  ls->premier  = NULL;
  ls->dernier  = NULL;
  ls->toString = toString;
  ls->comparer = comparer;
}

// créer et initialiser une liste symétrique
ListeS* creerListeSym (char* (*toString) (Objet*),
                                       int (*comparer) (Objet*, Objet*)) {
  ListeS* ls = new ListeS();
  initListeSym (ls, toString, comparer);
  return ls;
}

ListeS* creerListeSym() {
  return creerListeSym (toChar, comparerCar);  // par défaut
}

// la liste est-elle vide ?
booleen  listeVide (ListeS* ls) {
  return ls->premier == NULL;
}

// insérer "objet" en fin de la liste symétrique ls
void insererEnFinDeListeSym (ListeS* ls, Objet* objet) {
  Element* nouveau = creerElement();
  nouveau->reference = objet;
  nouveau->suivant = NULL;
  if (listeVide(ls)) {    // liste symétrique vide
    nouveau->precedent = NULL;
    ls->premier        = nouveau;
  } else {
    nouveau->precedent   = ls->dernier;
    ls->dernier->suivant = nouveau;
  }
  ls->dernier = nouveau;
}

// fournir un pointeur sur le premier élément de la liste
Element* premier (ListeS* ls) {
  return ls->premier;
}

// fournir un pointeur sur le dernier élément de la liste
Element* dernier (ListeS* ls) {
  return ls->dernier;
}

// fournir un pointeur sur le suivant de "elt"
Element* suivant (Element* elt) {
  return elt==NULL ? NULL : elt->suivant;
}

// fournir le précédent de "elt"
Element* precedent (Element* elt) {
  return elt==NULL ? NULL : elt->precedent;
}
```

Les parcours de listes symétriques :

```
// parcourir du premier vers le dernier
void parcoursListeSym (ListeS* ls, void (*f) (Objet*)) {
  if (listeVide(ls)) {
    printf ("Liste symétrique vide\n");
  } else {
    Element* ptc = premier (ls);
    while (ptc != NULL) {
      f (ptc->reference);
      ptc = suivant (ptc);
    }
  }
}

// parcours inverse : du dernier vers le premier
void parcoursListeSymI (ListeS* ls, void (*f) (Objet*)) {
  if (listeVide(ls)) {
    printf ("Liste symétrique vide\n");
  } else {
    Element* ptc = dernier (ls);
    while (ptc != NULL) {
      f (ptc->reference);
      ptc = precedent (ptc);
    }
  }
}

// chercher un pointeur sur l'élément contenant "objet" de la liste ls
static Element* chercherElement (ListeS* ls, Objet* objet) {
  booleen  trouve = faux;
  Element* ptc    = premier (ls);
  while ( (ptc != NULL) && !trouve ) {
    trouve = ls->comparer (objet, ptc->reference) == 0;
    if (!trouve) ptc = suivant (ptc);
  }
  return trouve ? ptc : NULL;
}

// chercher un pointeur sur l'objet "objet" de la liste ls
Objet* chercherObjet (ListeS* ls, Objet* objet) {
  Element* ptc = chercherElement (ls, objet);
  return ptc==NULL ? NULL : ptc->reference;
}
```

La fonction *void extraireListeSym (ListeS* ls, Element* extrait) ;* extrait l'élément pointé par *extrait* de la liste symétrique ls. Dans le cas général où l'élément à extraire se trouve entre deux autres éléments (donc pas en début ou fin de liste), on peut facilement définir un pointeur sur le précédent et un pointeur sur le suivant comme l'indique la Figure 40, et en conséquence, modifier le pointeur *precedent* du suivant et le pointeur *suivant* du précédent.

```
// retirer l'élément "extrait" de la liste symétrique ls;
// plus besoin d'avoir un pointeur sur le précédent
static void extraireListeSym (ListeS* ls, Element* extrait) {
  if ( (ls->premier==extrait) && (ls->dernier==extrait) ) {
    // suppression de l'unique élément de ls
    ls->premier = NULL;
```

```
    ls->dernier = NULL;
  } else if (ls->premier == extrait) {
    // suppression du premier de la liste ls
    ls->premier->suivant->precedent = NULL;
    ls->premier = ls->premier->suivant;

  } else if (ls->dernier == extrait) {
    // suppression du dernier de la liste ls
    ls->dernier->precedent->suivant = NULL;
    ls->dernier = ls->dernier->precedent;

  } else {
    // suppression de extrait entre 2 éléments non nuls
    extrait->suivant->precedent = extrait->precedent;
    extrait->precedent->suivant = extrait->suivant;
  }
}

void extraireListeSym (ListeS* ls, Objet* objet) {
  Element* element = chercherElement (ls, objet);
  if (element != NULL) extraireListeSym (ls, element);
}
```

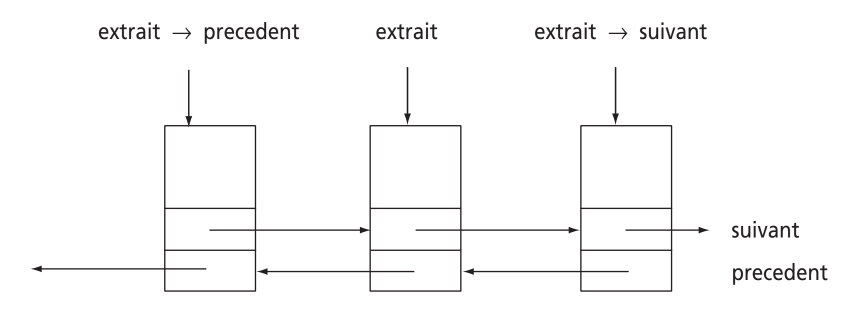


Figure 40 Extraction dans une liste symétrique.

2.8.3 Utilisation du module des listes symétriques

Le programme principal suivant est un simple programme de test du module des listes symétriques. La fonction *parcoursListeSym()* parcourt la liste en énumérant les éléments du premier vers le dernier ; la fonction *parcoursListeSymI()* énumère du dernier vers le premier ; il n'y a pas de tri à faire. La fonction *chercherObjet()* fournit un pointeur sur un objet de la liste à partir de son nom. Voir le type Personne, § 2.4.1, page 51.

```
/* pplistesym.cpp */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "listesym.h"
#include "mdtypes.h"

int menu () {
  printf ("\n\nLISTES SYMETRIQUES\n\n");
  printf ("0 - Fin\n");
```

```
  printf ("1 - Initialisation\n");
  printf ("2 - Insertion en fin de liste\n");
  printf ("3 - Parcours de liste\n");
  printf ("4 - Parcours inverse de liste\n");
  printf ("5 - Suppression d'un élément\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod);
  printf ("\n");
  return cod;
}

void main () {
  ListeS* ls = creerListeSym();
  booleen fini = faux;

  while (!fini) {
    switch (menu()) {
    case 0:
      fini = vrai;
      break;

    case 1: // initialisation de ls
      initListeSym (ls, toStringPersonne, comparerPersonne);
      break;

    case 2 : { // insertion d'un élément
      Personne* pers = creerPersonne();
      insererEnFinDeListeSym (ls, pers);
      } break;

    case 3: // parcours de liste
      parcoursListeSym (ls, ecrirePersonne);
      break;

    case 4: // parcours du dernier vers le premier
      parcoursListeSymI (ls, ecrirePersonne);
      break;

    case 5 : { // extraction d'un objet à partir de son nom
      printf ("Nom à extraire ? ");
      ch15 nom; scanf ("%s", nom);
      Personne* cherche = creerPersonne (nom, "?");
      Personne* ptc = (Personne*) chercherObjet (ls, cherche);
      if (ptc == NULL) {
        printf ("%s inconnu\n", nom);
      } else {
       ecrirePersonne (ptc);  // toutes les caractéristiques (nom, prénom)
        extraireListeSym (ls, ptc);
      }
      } break;
    }  // switch
  }  // while
}
```

Le programme utilisateur (*pplistesym.cpp*) n'utilise que les appels de fonctions de l'interface et n'accède pas directement aux informations de la tête de liste. D'autres fonctions pourraient être ajoutées au module sur les listes symétriques de façon à faciliter le travail de l'utilisateur de ce module.

2.9 ALLOCATION CONTIGUË

2.9.1 Allocation - désallocation en cas d'allocation contiguë

L'allocation dynamique en liste peut être simulée par une allocation contiguë (réserver un tableau ou de l'espace secondaire (fichiers sur disque) de taille définie) et une gestion *par le programmeur* de l'espace alloué (dans les applications où l'information est volatile : nombreux ajouts et retraits).

Allocation : il s'agit d'obtenir un nouvel emplacement pour créer un élément.

Désallocation : il s'agit de libérer un élément, la place mémoire (centrale ou secondaire) devenant disponible pour une éventuelle réutilisation ultérieure.

Soit la liste : Duroc - Durand - Dufour - Dupond. On examine plusieurs méthodes pour gérer l'espace alloué à l'application.

2.9.1.a Allocation séquentielle avec ramasse-miettes

Pour allouer, on choisit le premier libre (repéré par *pLibre* mémorisé dans l'entrée 0 par exemple). Sur la Figure 41, la première entrée libre à allouer est en 5. Après allocation de l'entrée 5, le champ *occupé* de 5 est mis à 1 et le premier libre se trouve alors en 6. Pour désallouer une entrée, il suffit de mettre à 0 le champ *occupé* de l'entrée correspondante.

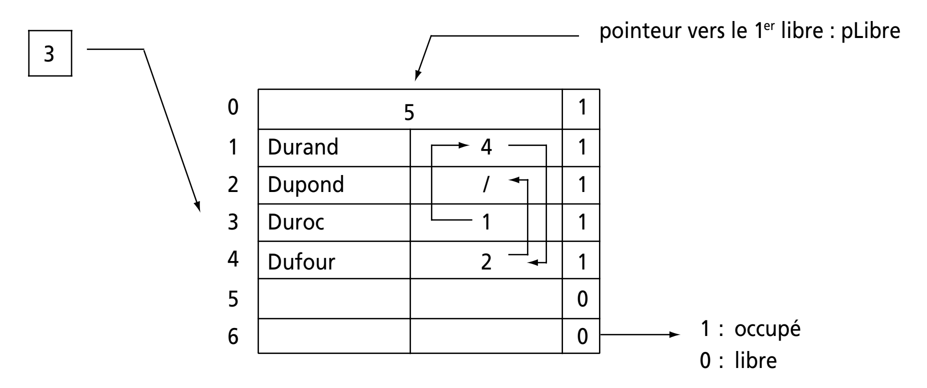


Figure 41 Gestion d'un espace mémoire (tableau ou fichier).

On ne retasse les éléments (ou on ne regénère le tableau ou le fichier) que s'il y a saturation de l'espace (appel du ramasse-miettes), les éléments libérés n'étant pas réutilisés tant qu'il reste de la place en fin de tableau ou de fichier. Les changements de valeur des suivants en cas de suppression d'un élément et retassement sont peu efficaces : il faut modifier tous les champs *suivant* supérieurs à l'entrée libérée. En cas de suppression d'un élément, on peut aussi recopier le dernier occupé à la place de l'élément supprimé, modifier en conséquence les listes incluant l'élément déplacé et détruire le dernier. De cette façon, les éléments restent consécutifs. Le premier élément peut avoir le numéro 0 ou 1 ; l'absence de suivant (Nil) peut être

notée –1 ou 0. S'il s'agit d'un fichier, les nombreuses modifications d'enregistrements font que la méthode est inappropriée car lente.

2.9.1.b Marqueur par élément (table d'occupation)

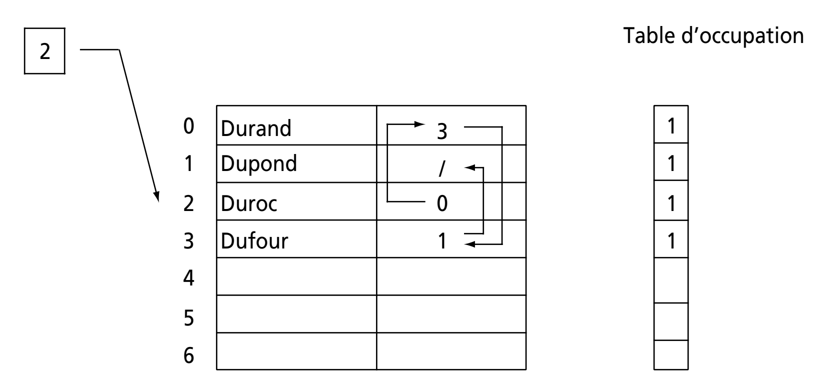


Figure 42 Gestion d'espace secondaire (disque) avec table d'occupation.

Pour un fichier, il est beaucoup plus efficace de réunir les marqueurs d'occupation dans un fichier à part dit "table d'occupation" qui est chargée en mémoire centrale avant l'utilisation du fichier. On évite ainsi les nombreux accès disque pour allouer ou désallouer. Allouer consiste à parcourir la table d'occupation à la recherche d'un élément binaire 0 qui est mis à 1. Le rang de l'élément binaire indique le rang de l'entrée libre à allouer dans le tableau ou le fichier en accès direct. Désallouer consiste à mettre l'élément binaire de la table d'occupation à 0 (voir Figure 42).

2.9.1.c Éléments libres en liste (allocation contiguë)

On peut aussi faire une liste des éléments libres. Pour trouver une place pour un nouvel élément, il suffit d'extraire le premier de la liste libre et d'y ranger les informations caractéristiques de l'application. Pour détruire un élément devenu inutile, il suffit de l'insérer en tête de la liste libre (voir Figure 43).

Initialiser	consiste à insérer tous les éléments dans la liste libre
Allouer	consiste à extraire un élément en tête de la liste libre
Libérer	consiste à insérer l'élément à libérer en tête de la liste libre

Le dernier élément libéré est le premier à être à nouveau alloué.

2.9.2 Exemple des polynômes en allocation contiguë avec liste libre

La gestion de polynômes d'une variable réelle a déjà été traitée en utilisant l'allocation dynamique (voir § 2.4.3, page 55). La mémorisation peut aussi se faire en utilisant l'allocation contiguë. Le programmeur doit définir un espace pour la mémorisation des différents polynômes et doit le gérer lui-même, c'est-à-dire connaître en permanence ce qui est occupé et ce qui est libre. Il doit pouvoir allouer une nouvelle entrée du tableau pour y loger un monôme ou désallouer une entrée

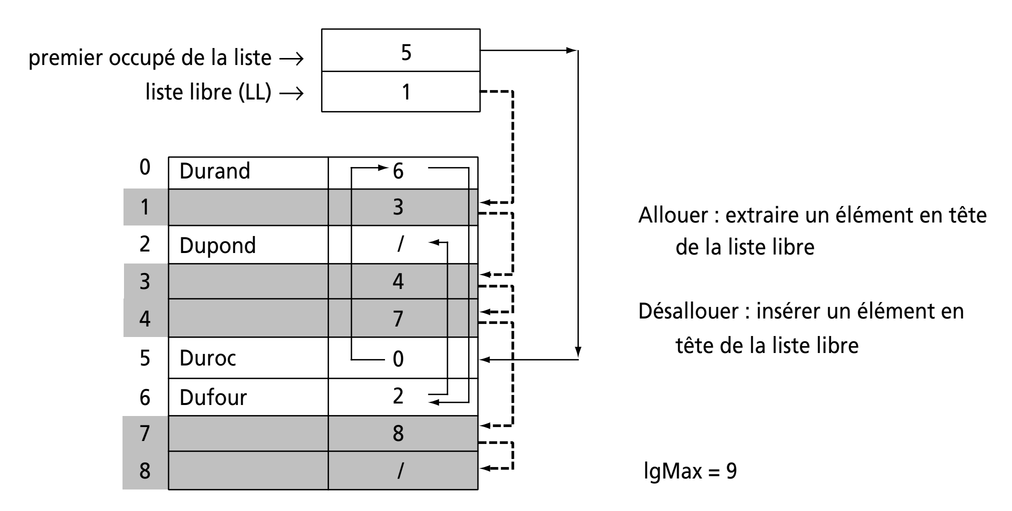


Figure 43 Gestion d'espace mémoire avec une liste libre.

devenue libre suite à la destruction d'un monôme. La Figure 44 montre la mémorisation des polynômes suivants :

$$A = 3x^5 + 2x^3 + 1$$

$$B = 6x^5 – 5x^4 – 2x^3 + 8\ x^2$$

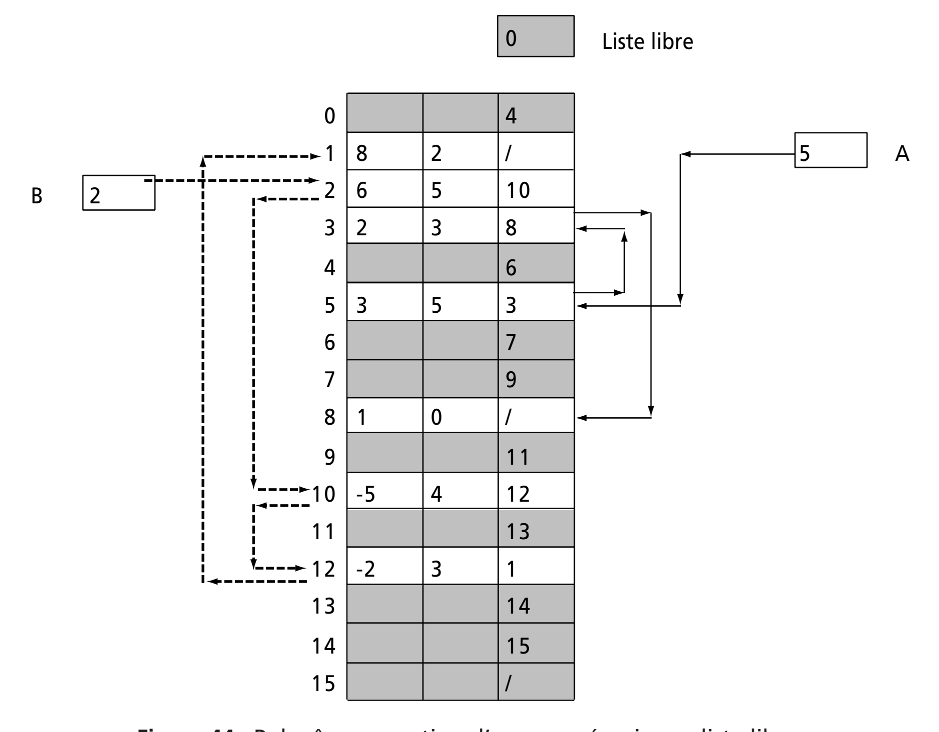


Figure 44 Polynômes : gestion d'espace mémoire en liste libre.

L'allocation d'un nouveau monôme consiste à extraire un élément de la liste libre. Sur le schéma, c'est l'entrée 0 qui sera allouée en cas de nouvelle allocation. La désallocation d'un monôme consiste à insérer l'entrée en tête de la liste libre.

Déclaration en cas d'allocation sous forme de tableaux :

```
#define NMAX 16
typedef struct {
  double coefficient;
  int    exposant;
  int    suivant;       // indice du suivant
} Monome;

Monome monome [NMAX];   // l'espace à gérer
```

Dans les fonctions de gestion en allocation dynamique de listes p->suivant, par exemple, s'écrit monome[p].suivant en allocation contiguë. Dès lors que les informations sont en mémoire centrale, l'allocation dynamique est plus simple à mettre en œuvre, le système d'exploitation se chargeant de gérer l'espace alloué.

2.9.3 Exemple de la gestion des commandes en attente

Il s'agit de gérer les commandes en attente d'une société. Les commandes en attente concernent des clients et des articles. On peut schématiser les entités et les relations sur un modèle conceptuel des données (MCD) comme indiqué sur la Figure 45.

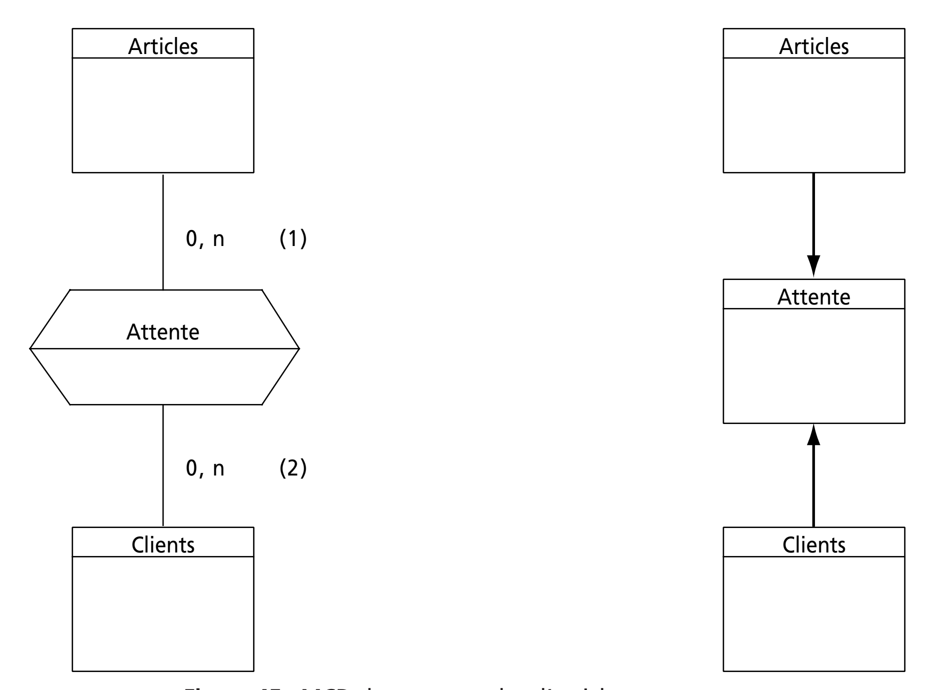


Figure 45 MCD des commandes d'articles en attente.

(1) un article donné est en attente pour de 0 à n clients
(2) un client donné attend de 0 à n articles

2.9.3.a Exemples d'opérations envisageables

Initialisation des fichiers : le fichier Attente est géré avec une liste libre (voir Figure 46) car il est très volatil. Il y a de nombreux ajouts et retraits. On peut espérer que les commandes en attente ne resteront pas trop longtemps en attente.

Mise en attente des commandes suivantes (article, date, quantité, client) :

–Téléviseur	03/07/..	3	Dupond
–Radio	10/08/..	1	Dupond
–Chaîne hi-fi	02/09/..	5	Dupond
–Téléviseur	12/09/..	10	Durand
–Chaîne hi-fi	13/09/..	7	Durand

Interrogations possibles :

–liste des articles en attente pour un client

–liste des clients en attente pour un article

–liste des envois à faire suite au réapprovisionnement d'un article (suppression des commandes en attente satisfaites)

2.9.3.b Description des fichiers

Articles :

- nom de l'article
- la quantité en stock (1)
- la première commande en attente pour cet article (2)
- la dernière commande en attente pour cet article (3)

Clients :

- nom du client
- la première commande en attente pour ce client (1)
- la dernière commande en attente pour ce client (2)

Attente :

- la date de la commande
- la quantité commandée (1)

- le numéro de l'article concerné (2)
- l'entrée suivante pour le même article (3)
- l'entrée précédente pour le même article (4)

- l'entrée suivante pour le même client (5)
- l'entrée précédente pour le même client (6)

• le numéro du client concerné, ou si l'entrée est libre, le numéro de la prochaine entrée libre (7)

	Articles			
0				
1	Radio	0	6	6
2				
3	Téléviseur	0	3	8
4				
5				
6				
7				
8	Magnétoscope	25	/	/
9				
10	Chaîne hi-fi	0	12	14
11				
	nomArt	(1)	(2)	(3)

	Clients		
0			
1			
2	Dupond	3	12
3			
4			
5	Durand	8	14
6	Dufour	/	/
7			
8			
	nomCli	(1)	(2)

	Attente							
0								1
1								2
2								4
3	03/07/..	3	3	8	-1	6	-1	2
4								5
5								7
6	10/08/..	1	1	-1	-1	12	3	2
7								9
8	12/09/..	10	3	-1	3	14	-1	5
9								10
10								11
11								13
12	02/09/..	5	10	14	-1	-1	6	2
13								15
14	13/09/..	7	10	-1	12	-1	8	5
15								16
16								17
17								/
	date	(1)	(2)	(3)	(4)	(5)	(6)	(7)

Figure 46 Commandes en attente : utilisation de listes symétriques.

2.9.3.c Explications de l'exemple de la Figure 46

Le fichier Attente est géré à l'aide d'une liste libre ce qui permet la réutilisation des entrées devenues disponibles suite à un réapprovisionnement par exemple. Sur l'exemple, la tête de liste est mémorisée dans l'entrée 0 ; les entrées disponibles en colonne (7) de Attente sont 1, 2, 4, 5, 7, 9, etc.

Les éléments sont chaînés à l'aide de listes symétriques ce qui facilite l'extraction d'un élément des listes auxquelles il appartient en ne connaissant que son numéro d'entrée.

Pour *Téléviseur* (article 3) par exemple, le premier article en attente est à l'entrée 3 du fichier Attente, le dernier à l'entrée 8 (champs 2 et 3 de Articles). Les champs (3) et (4) de Attente contiennent les pointeurs *suivant* et *precedent* des commandes en attente pour un article donné. Le pointeur (7) de Attente indique, si l'enregistrement est occupé, le numéro du client concerné par cette commande en attente ; on peut donc retrouver toutes les caractéristiques du client.

De même, pour *Dupond* (client 2), la première commande en attente est en 3 et la dernière en 12 (champs 1 et 2 de Clients). Les champs (5) et (6) de Attente contiennent les pointeurs *suivant* et *precedent* des commandes en attente pour un client donné. Le pointeur (2) de Attente indique le numéro de l'article concerné par la commande en attente ; on peut donc retrouver les caractéristiques du produit et en particulier son nom.

Cette structure est surtout valable si les fichiers sont importants. On peut très rapidement à partir du numéro du client, retrouver ses commandes en attente avec toutes leurs caractéristiques sans opérer de sélection ou de tri. Il suffit de suivre les pointeurs. La réciproque est aussi vraie : retrouver pour un article les clients en attente. En cas de réapprovisionnement d'un article, il suffit également de suivre les pointeurs pour satisfaire les commandes en souffrance pour cet article et libérer les entrées de attente devenues libres.

2.9.3.d Le fichier d'en-tête de la gestion en liste libre du fichier Attente

Ce module effectue la gestion de la liste libre du fichier Attente. *lireDAtt()* et *ecrireDAtt()* permettent un accès direct à un enregistrement du fichier Attente à partir de son numéro. *lireTeteListe()* et *ecrireTeteListe()* permettent de lire ou de modifier la valeur de la tête de la liste libre (mémorisée dans l'enregistrement 0). *allouer()* fournit un numéro d'enregistrement libre. *liberer()* réinsère un enregistrement dans la liste libre. *initAtt()* initialise la liste libre du fichier Attente.

```
/* attente.h
   Gestion du fichier Attente utilisant une liste libre */

#ifndef ATTENTE_H
#define ATTENTE_H

#define NBENR 20    // nombre d'enregistrements dans le fichier
#define NILE  -1
```

```
typedef char ch8 [9];
typedef struct {
  int occupe;          // enregistrement occupé
  ch8 date;            // date de la commande
  int qt;              // quantité commandée
  int numArt;          // numéro de l'article
  int artSuivant;      // article suivant
  int artPrecedent;    // article précédent
  int cliSuivant;      // client suivant
  int cliPrecedent;    // client précédent
  int cliOuLL;         // numéro de Client ou Liste Libre
} Attente;

void lireDAtt        (int n, Attente* enrAtt);
void ecrireDAtt      (int n, Attente* enrAtt);
int  lireTeteListe   ();
void ecrireTeteListe (int    listLibre);
int  allouer         ();
void liberer         (int    nouveau);
void initAtt         (char*  nom);
void fermerAtt       ();

#endif
```

2.9.3.e Le module de gestion en liste libre du fichier Attente

Le module correspond aux fonctions définies dans le fichier d'en-tête précédent.

```
/* attente.cpp  Module de Gestion du fichier attente
   (utilisation d'une liste libre et de listes symétriques) */

#include <stdio.h>
#include <stdlib.h>      // exit
#include "attente.h"

FILE* fr;  // fichier relatif Attente

// lire Directement dans le fichier attente l'enregistrement n,
// et le mettre dans la structure pointée par attente
void lireDAtt (int n, Attente* attente) {
  fseek (fr, (long) n*sizeof (Attente), 0);
  fread (attente, sizeof (Attente), 1, fr);
}

// écrire Directement dans le fichier attente l'enregistrement n,
// à partir de la structure pointée par attente
void ecrireDAtt (int n, Attente* attente) {
  fseek  (fr, (long) n*sizeof (Attente), 0);
  fwrite (attente, sizeof (Attente), 1, fr);
}

// fournir la valeur de la tête de liste
int lireTeteListe () {
  Attente attente;
  lireDAtt (0, &attente);
  return attente.cliOuLL;
}

// écrire la valeur de listLibre dans la tête de liste
void ecrireTeteListe (int listLibre) {
  Attente attente;
```

```
    attente.cliOuLL = listLibre;
    ecrireDAtt (0, &attente);
  }

  // fournir le premier libre de la liste libre,
  // ou NILE si la liste libre est vide
  int allouer () {
    Attente attente;
    int nouveau = lireTeteListe();
    if (nouveau != NILE) {
      lireDAtt (nouveau, &attente);
      ecrireTeteListe (attente.cliOuLL);
    }
    return nouveau;
  }

  // ajouter nouveau en tête de la liste libre
  void liberer (int nouveau) {
    Attente attente;
    attente.occupe  = 0;
    attente.cliOuLL = lireTeteListe();
    ecrireDAtt      (nouveau, &attente);
    ecrireTeteListe (nouveau);
  }

  // initialiser le fichier relatif, et la liste libre
  void initAtt (char* nom) {
    Attente attente;

    fr = fopen (nom, "wb+");
    if (fr==NULL) { perror ("fichier inconnu : "); exit (1); }

    ecrireTeteListe (1);
    attente.occupe = 0;

    for (int i=1; i<NBENR-1; i++) {
      attente.cliOuLL = i+1;
      ecrireDAtt (i, &attente);
    }
    attente.cliOuLL = NILE;  // le dernier enregistrement
    ecrireDAtt (NBENR, &attente);

    // pour avoir le schéma du cours
    // soit la liste libre 3, 6, 12, 8, 14, -1
    ecrireTeteListe (3);
    attente.cliOuLL = 6;  ecrireDAtt (3,  &attente);
    attente.cliOuLL = 12; ecrireDAtt (6,  &attente);
    attente.cliOuLL = 8;  ecrireDAtt (12, &attente);
    attente.cliOuLL = 14; ecrireDAtt (8,  &attente);
    attente.cliOuLL = -1; ecrireDAtt (14, &attente);
  }

  void fermerAtt () {
    fclose (fr);
  }
```

2.9.3.f Le programme de gestion des commandes en attente

Des parties du programme principal de la gestion des commandes en attente sont données pour illustrer l'utilisation d'enregistrements chaînés et l'allocation en liste libre. D'autres parties sont laissées en exercice.

```
/* ppattente.cpp
   Fichiers des commandes en attente
   avec liste libre et listes symétriques */

#include <stdio.h>
#include <string.h>    // strcpy
#include <stdlib.h>    // exit
#include "attente.h"

typedef int  booleen;
#define faux 0
#define vrai 1
typedef char ch15 [16];

// les articles
typedef struct {
  ch15 nomArt;
  int  qts;         // quantité en stock
  int  premier;     // liste symétrique pour un article
  int  dernier;
} Article;

// les clients
typedef struct {
  ch15 nomCli;
  int  premier;     // liste symétrique pour un client
  int  dernier;
} Client;

FILE* fa;  // fichier des articles
FILE* fc;  // fichier des clients

// lire directement l'enregistrement n de fa
void lireDArt (int n, Article* article) {
  fseek (fa, (long) n*sizeof (Article), 0);
  fread (article,     sizeof (Article), 1, fa);
}

// lire directement l'enregistrement n de fc
void lireDCli (int n, Client* client) {
  fseek (fc, (long) n*sizeof (Client), 0);
  fread (client,      sizeof (Client), 1, fc);
}

// écrire directement l'enregistrement n de fa
void ecrireDArt (int n, Article* article) {
  fseek  (fa, (long) n*sizeof (Article), 0);
  fwrite (article,     sizeof (Article), 1, fa);
}

// écrire directement l'enregistrement n de fc
void ecrireDCli (int n, Client* client) {
  fseek  (fc, (long) n*sizeof (Client), 0);
  fwrite (client,      sizeof (Client), 1, fc);
}

// liste des articles en attente pour le client n
void listerArt (int n) {
  Attente  attente;
  Article  article;
  Client   client;

  lireDCli (n, &client);
```

```
  printf ("\nListe des articles en attente pour %s\n", client.nomCli);
  int ptc = client.premier;

  while (ptc != NILE) {
    lireDAtt (ptc, &attente);
    printf   ("%10s ", attente.date);
    lireDArt (attente.numArt, &article);
    printf   ("%s\n", article.nomArt);
    ptc = attente.cliSuivant;
  }
  printf ("\n");
}

// liste des clients en attente de l'article n
void listerCli (int n) {
  Attente  attente;
  Article  article;
  Client   client;

  lireDArt (n, &article);
  printf ("\nListe des clients en attente de %s\n", article.nomArt);
  int ptc = article.premier;

  while (ptc != NILE) {
    lireDAtt (ptc, &attente);
    printf   ("%10s", attente.date);
    lireDCli (attente.cliOuLL, &client);
    printf   (" %s\n", client.nomCli);
    ptc = attente.artSuivant;
  }
  printf ("\n");
}

// initialiser la nième entrée du fichier article
void initArt (char* nomArt, int n, int qts) {
  Article article;
  strcpy (article.nomArt, nomArt);
  article.qts = qts;
  article.premier = NILE;
  article.dernier = NILE;
  ecrireDArt (n, &article);
  //printf ("initArt %s\n",  nomArt);
}


// initialiser la nième entrée du fichier client
void initCli (char* nomCli, int n) {
  Client client;
  strcpy (client.nomCli, nomCli);
  client.premier = NILE;
  client.dernier = NILE;
  ecrireDCli (n, &client);
  //printf ("initCli %s\n",  nomCli);
}

// mise en attente de qt article numArt pour le client numCli
void mettreEnAttente (int qt, int numArt, int numCli, char* date) {
  // à faire en exercice
}
```

```
// extraire l'entrée n des listes de Attente
void extraire (int n) {
// à faire en exercice
}

// réapprovisionnement de qtr articles de numéro na
void reappro (int na, int qtr) {
// à faire en exercice
}

void main () {
// à faire en exercice
}
```

Exemple de résultats à partir de la Figure 46.

```
Fichier attente
 3   03/07/..  3,  3  8 -1 :  6 -1  2
 6   10/08/..  1,  1 -1 -1 : 12  3  2
 8   12/09/.. 10,  3 -1  3 : 14 -1  5
12   02/09/..  5, 10 14 -1 : -1  6  2
14   13/09/..  7, 10 -1 12 : -1  8  5

Fichier Article
1 Radio             6   6
3 Téléviseur        3   8
8 Magnétoscope     -1  -1
10 Chaîne hi-fi     12  14

Fichier client
2 Dupond            3  12
5 Durand            8  14
6 Dufour           -1  -1



Liste des articles en attente pour Durand
  12/09/.. Téléviseur
  13/09/.. Chaîne hi-fi


Liste des clients en attente de Téléviseur
  03/07/.. Dupond
  12/09/.. Durand
```

Exercice 12 - Commande en attente

• Écrire la fonction *void **mettreEnAttente** (int qt, int numArt, int numCli, char* date) ;* qui insère une nouvelle commande en attente pour l'article numArt et le client numCli à la date *date*. qt indique la quantité commandée.

• Écrire la fonction *void* ***extraire*** *(int n) ;* qui extrait l'enregistrement n des deux listes symétriques auxquelles il appartient, et libère (désalloue) cet enregistrement.

• Écrire la fonction *void* ***reappro*** *(int na, int qtr) ;* qui lance les commandes en attente suite à un réapprovisionnement de qtr articles de numéro na. Les commandes sont satisfaites dans l'ordre chronologique. La dernière commande peut n'être satisfaite qu'en partie.

Exercice 13 - Les cartes à jouer

On dispose du module de gestion des listes simples décrit dans le fichier d'en-tête *liste.h* et définissant les diverses fonctions opérant sur les listes. On veut simuler par programme la distribution de cartes à jouer pour un jeu de 52 cartes. Il y a 4 couleurs de cartes (numérotées de 1 à 4), et 13 valeurs de cartes par couleur (numérotées de 1 à 13). On définit le fichier d'en-tête *cartes.h* suivant.

```
/* cartes.h */

#ifndef CARTE_H
#define CARTE_H

#include "liste.h"

typedef struct {
  int   couleur;
  int   valeur;
} Carte;

typedef Liste       PaquetCarte;
typedef PaquetCarte tabJoueur [4];

void    insererEnFinDePaquet (PaquetCarte* p,  int couleur, int valeur);
void    listerCartes         (PaquetCarte* p);
void    creerTas             (PaquetCarte* p);
Carte*  extraireNieme        (PaquetCarte* p,  int n);
void    battreLesCartes      (PaquetCarte* p,  PaquetCarte* paquetBattu);
void    distribuerLesCartes  (PaquetCarte* p,  tabJoueur    joueur);

#endif
```

Écrire les fonctions suivantes du module *cartes.cpp* :

• *void insererEnFinDePaquet (PaquetCarte* p, int couleur, int valeur) ;* qui insère une carte de couleur et de valeur données en fin du paquet p.

• *void listerCartes (PaquetCarte* p) ;* qui liste la couleur et la valeur des cartes du paquet de cartes p.

• *void creerTas (PaquetCarte* p) ;* qui crée un paquet de cartes p contenant les cartes dans l'ordre couleur 1 pour les 13 cartes, puis couleur 2 pour les 13 suivantes, etc., soit en tout 52 cartes.

• *Carte* extraireNieme (PaquetCarte* p, int n) ;* qui extrait la nième carte du paquet p et fournit un pointeur sur la carte extraite.

• *void battreLesCartes (PaquetCarte* p, PaquetCarte* paquetBattu) ;* qui crée à partir du *PaquetCarte p* contenant 52 cartes, un nouveau *PaquetCarte paquetBattu* résultat. On extrait aléatoirement une carte du paquet p pour l'insérer en fin de *paquetBattu* jusqu'à ce que p soit vide. Utiliser la fonction *rand()* de génération de nombre aléatoire.

• void distribuerLesCartes (PaquetCarte* p, tabJoueur joueur) ; qui distribue jusqu'à épuisement de p (52 cartes), une carte à chacun des 4 joueurs.

Écrire le programme principal ppcartes.cpp qui génère un paquet de 52 cartes, les affiche, les bat, les affiche à nouveau, les distribue aux 4 joueurs, et affiche la poignée de chaque joueur.

Exercice 14 - Polynômes complexes

On veut utiliser des polynômes d'une variable complexe. Les polynômes sont mémorisés dans des listes ordonnées décroissantes suivant l'exposant. Chaque objet de la liste est composé d'un nombre complexe **z** et d'un entier **puissance** (puissance de z du monôme). On dispose du module sur les complexes défini précédemment lors de l'exercice 6, page 32 (fichier d'en-tête : *complex.h*) et du module sur les listes ordonnées (voir § 2.3.8, page 48 : fichier d'en-tête *liste.h*).

Soit le fichier d'en-tête *polynome.h* suivant :

```
/* polynome.h */

#ifndef POLYNOME_H
#define POLYNOME_H

#include "complex.h"
#include "liste.h"

typedef struct {
  Complex  z;
  int      puissance;
} Monome;

typedef Liste Polynome;

Polynome* creerPolynome  ();
void      creerMonome    (Polynome* p, Complex z, int puis);
Polynome* lirePolynome   ();
void      ecrirePolynome (Polynome* p);
Complex   valeurPolynome (Polynome* p, Complex z);

#endif
```

Écrire les fonctions (fichier *polynome.cpp*) réalisant les opérations suivantes sur les polynômes d'une variable complexe :

- créer un polynôme vide (liste ordonnée),
- créer un monôme et l'ajouter au polynôme ordonné,
- lire un polynôme complexe sur l'entrée standard (clavier),
- écrire un polynôme complexe,
- calculer la valeur d'un polynôme complexe pour une valeur de z donnée.

Écrire un programme d'application (fichier *pppolynomecomplex.cpp*) réalisant :

- la lecture de polynômes complexes,
- l'écriture de polynômes complexes,
- le calcul de la valeur d'un polynôme complexe pour une valeur complexe z donnée,
- le calcul de la valeur d'un produit de polynômes complexes pour un z donné,
- le calcul de la valeur d'un quotient de polynômes complexes pour un z donné.

Exemples de résultats attendus pour les polynômes en z suivants :

$p1 = (3+3i)\ z^3 + (2+2i)\ z^2 + (1+i)\ z$
$p2 = (1+i)\ z^3 + (3+3i)\ z^2 + (2+2i)\ z$

```
Liste du polynome p1 :  (3.00 + 3.00 i) z** 3
    +  (2.00 + 2.00 i) z** 2  +  (1.00 + 1.00 i) z** 1

Liste du polynome p2 :  (1.00 + 1.00 i) z** 3
    +  (3.00 + 3.00 i) z** 2  +  (2.00 + 2.00 i) z** 1

z1             =  (  0.50 +  1.00 i )
p1(z1)         =  ( -7.38 + -2.87 i )
p2(z1)         =  ( -7.38 +  2.13 i )
p1(z1)*p2(z1) =  ( 60.50 +  5.53 i )
p1(z1)/p2(z1) =  (  0.82 +  0.63 i )
```

2.10 RÉSUMÉ

Les listes sont des structures de données permettant de gérer facilement des ensembles de valeurs (ordonnées ou non) de taille a priori inconnue. L'utilisation de listes d'éléments alloués dynamiquement facilite les insertions et les suppressions d'éléments lorsque l'information évolue (apparaît et disparaît). Si on insère les éléments suivant un critère d'ordre, on peut facilement obtenir une liste triée. Un tableau au contraire nécessite une borne supérieure indiquant le nombre maximum d'éléments mémorisables. Les retraits d'éléments d'un tableau et la réutilisation de l'espace libéré sont par contre plus difficiles à gérer. Cependant, la plupart des traitements sur les listes sont séquentiels, ce qui veut dire que pour accéder à un élément,

il faut consulter les précédents. On ne peut pas se positionner directement sur un élément. Si la liste contient de nombreux éléments, les traitements risquent de s'allonger.

Le module de traitement des listes présenté dans ce chapitre est très général et peut être facilement réutilisé dans de nombreuses applications comme l'indiquent les divers exemples traités. La liste est un type abstrait de données que l'on met en œuvre en respectant les prototypes de l'interface du module liste.h. Il existe plusieurs variantes des listes (circulaires, symétriques) facilitant un des aspects du traitement des listes (l'extraction par exemple pour les listes symétriques).

L'allocation contiguë permet de regrouper les informations de la liste dans un même espace contigu en mémoire centrale ou secondaire. Cependant le programmeur doit gérer lui-même cet espace et être en mesure d'allouer une place pour un élément ou de libérer une place qui pourra être réutilisée par la suite. Sauf raisons très particulières, il vaut mieux, en mémoire centrale, utiliser l'allocation dynamique et bénéficier ainsi de la gestion de mémoire faite par le système d'exploitation. Sur mémoire secondaire, l'utilisateur doit gérer son espace comme on l'a vu sur l'exemple des commandes en attente.

Chapitre 3

Les arbres

3.1 LES ARBRES N-AIRES

3.1.1 Définitions

Arbre : un arbre est une structure de données composée d'un ensemble de nœuds. Chaque nœud contient l'information spécifique de l'application et des pointeurs vers d'autres nœuds (d'autres sous-arbres).

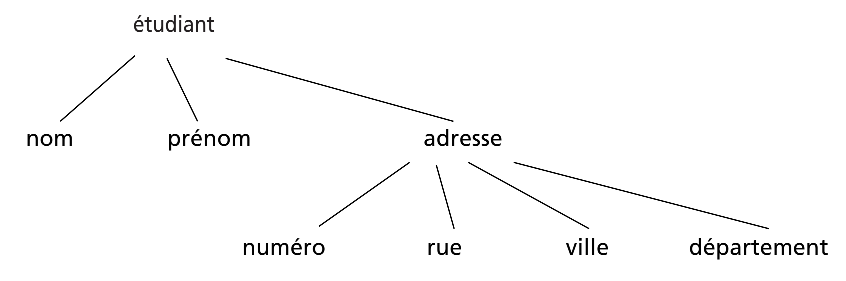


Figure 47 Un arbre n-aire.

L'arbre de la Figure 47 peut se noter :

(étudiant (nom, prénom, adresse (numéro, rue, ville, département))).

Feuilles : les nœuds ne pointant vers aucun autre nœud sont appelés feuilles (*nom, prénom, numéro, rue, ville, département* sont des feuilles).

Racine : il existe un nœud au niveau 1 qui n'est pointé par aucun autre nœud : c'est la racine de l'arbre (*étudiant* est la racine de l'arbre).

Niveau : le niveau de la racine est 1. Les autres nœuds ont un niveau qui est augmenté de un par rapport au nœud dont ils dépendent.

Hauteur (profondeur) d'un arbre : c'est le niveau maximum atteint (par la branche la plus longue). La hauteur du nœud *étudiant* est de 3.

Arbre ordonné : si l'ordre des sous-arbres est significatif, on dit que l'arbre est ordonné (arbre généalogique par exemple).

Arbre binaire : un arbre binaire est un type d'arbre ordonné tel que chaque nœud a au plus deux fils et quand il n'y en a qu'un, on précise s'il s'agit du fils droit ou du fils gauche.

Degré d'un nœud : on appelle degré d'un nœud, le nombre de successeurs de ce nœud.

Degré d'un arbre : si N est le degré maximum des nœuds de l'arbre, l'arbre est dit n-aire. Sur l'exemple, *adresse* a un degré 4. L'arbre est un arbre 4-aire.

Taille : c'est le nombre total de nœuds de l'arbre. La taille est de 8 sur l'exemple de la Figure 47.

Arbre binaire complet : c'est un arbre binaire de taille 2^k-1 (k étant le niveau des feuilles).

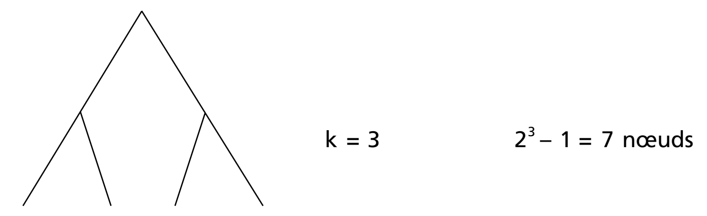


Figure 48 Un arbre complet.

Arbre binaire parfaitement équilibré : un arbre binaire est parfaitement équilibré si pour chaque nœud, les **nombres de nœuds** des sous-arbres gauche et droit diffèrent au plus d'un.

Arbre binaire équilibré : un arbre binaire est équilibré si pour chaque nœud, les **hauteurs** des sous-arbres gauche et droit diffèrent au plus d'un.

3.1.2 Exemples d'applications utilisant les arbres

a) Expression arithmétique : arbre binaire ordonné

Une expression arithmétique ayant des opérateurs binaires peut être schématisée sous la forme d'un arbre binaire. La Figure 49 représente l'expression arithmétique : ((a+b) * (c-d) – e).

L'arbre est ordonné : permuter 2 sous-arbres change l'expression.

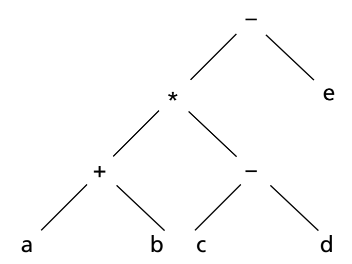


Figure 49 L'arbre d'une expression arithmétique.

b) Représentation de caractères

Soient à représenter les mots suivants : *mais, mars, mer, mon, sa, son, sel*. Les débuts communs peuvent n'être mémorisés qu'une seule fois sous forme d'un arbre de caractères. L'arbre peut aussi se noter sous la forme équivalente suivante : (m (a (is, rs), er, on), s (a, on, el)). En fait, il faut ajouter un caractère '*' en fin de chaque mot, pour pouvoir distinguer les mots sous-chaînes d'un mot plus long comme par exemple *ma* et *mars*.

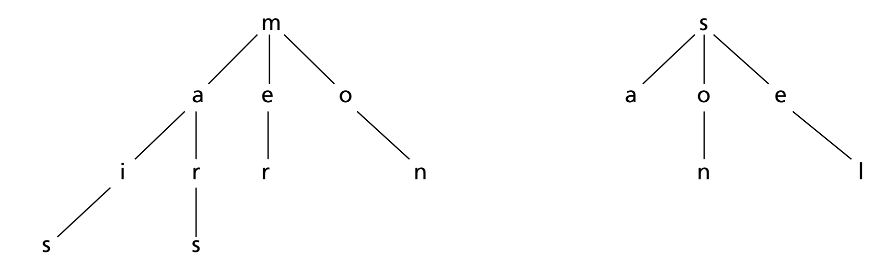


Figure 50 Un arbre de mots.

c) Structure d'une phrase : arbre n-aire ordonné

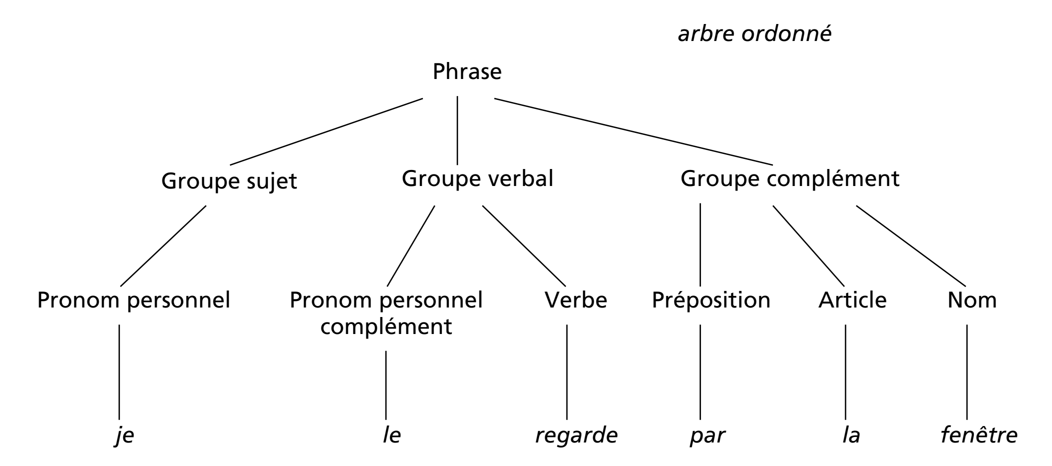


Figure 51 L'arbre syntaxique d'une phrase française.

Dans les traitements informatiques de la langue naturelle, on a souvent besoin de connaître la structure d'une phrase pour :

• traduire cette phrase d'une langue dans une autre langue, (exemple : « Mary was told that John left yesterday » devient « On a dit à Marie que Jean était parti hier »),

• prononcer la phrase sur synthétiseur de parole (une bonne intonation nécessite une certaine connaissance de la structure de la phrase),

• comprendre le sens de la phrase et agir en fonction de la commande donnée en langage naturel. Pour la commande d'un robot, on pourrait imaginer l'ordre suivant en langage naturel : mettre la sphère verte sur le cube rouge.

Dans certaines applications, le langage peut être contraint, c'est-à-dire limité par le vocabulaire de l'application et par les constructions syntaxiques acceptées qui doivent être conformes à la grammaire de l'application. Par contre, en synthèse de parole, le synthétiseur doit être capable de prononcer n'importe quel mot, nom propre ou abréviation.

d) Arbre généalogique : arbre n-aire

L'arbre généalogique suivant est un arbre de descendance. Julie a deux enfants : Jonatan et Gontran. Jonatan a trois enfants : Pauline, Sonia, Paul. Le degré de l'arbre est de 3 ; l'arbre est dit 3-aire ou n-aire d'une manière générale. Le degré de chaque nœud est variable puisqu'il dépend du nombre d'enfants de ce nœud. Julie est la racine de l'arbre.

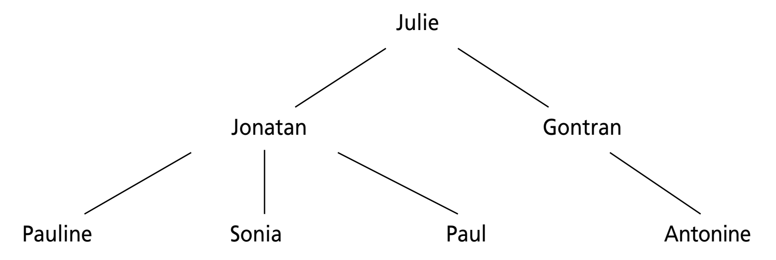


Figure 52 Un arbre généalogique.

e) Tournoi de tennis

Un tournoi se schématise sous la forme d'un arbre, les matchs se déroulant des feuilles vers la racine. Michel a battu Jérôme ; Gérard a battu Olivier. Les gagnants ont joué ensemble et c'est Michel qui a gagné.

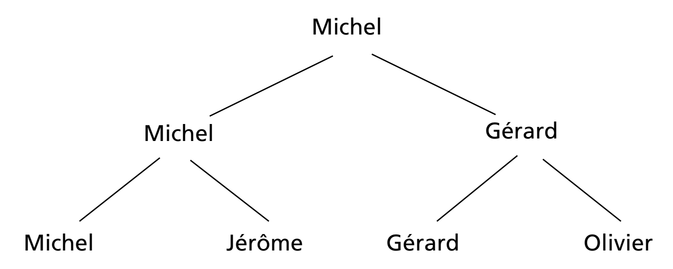


Figure 53 L'arbre d'un tournoi de tennis.

f) ou encore

• la structure d'un chapitre de cours est arborescente : le chapitre se découpe en sections et paragraphes.

• le répertoire des fichiers d'un système d'exploitation a une structure arborescente faite de sous-répertoires et de fichiers.

• l'interface graphique d'un logiciel est constituée de fenêtres et sous-fenêtres qui forment un arbre.

• la nomenclature d'un objet est un arbre : l'arbre des composants d'une voiture (moteur, carrosserie, sièges, etc.), de la structure de la Terre (continents, pays, etc.), du corps humain (tête, tronc, membres, etc.).

• la classification des espèces animales en vertébrés (poissons, batraciens, reptiles, oiseaux, mammifères) et invertébrés (arthropodes (crustacés, myriapodes, arachnides, insectes), vers ou mollusques) forme un arbre. De même pour la classification des espèces végétales en phanérogames (monocotylédones, dicotylédones) ou cryptogames (algues, champignons, lichens, mousses, fougères).

3.1.3 Représentation en mémoire des arbres n-aires

Il s'agit de mémoriser les arbres et leurs relations de dépendance père fils.

3.1.3.a Mémorisation par listes de fils

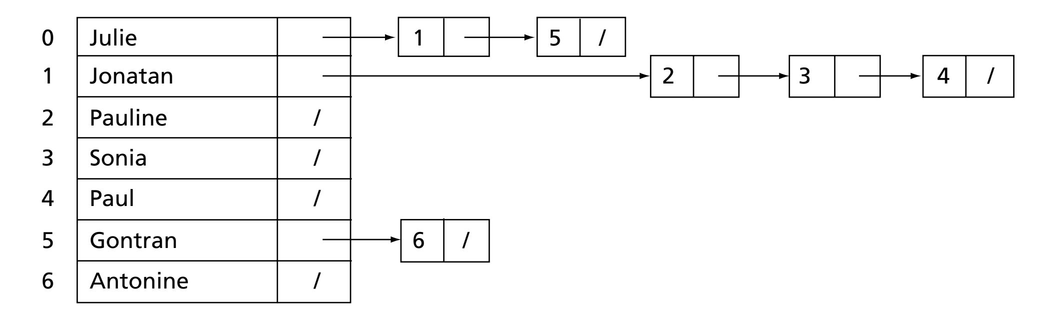


Figure 54 Mémorisation d'un arbre par listes de fils.

L'arbre de la Figure 52 peut se mémoriser comme l'indique la Figure 54. L'allocation est dans ce cas en partie contiguë et en partie dynamique. Avec cette structure de données, les insertions et les suppressions de nœuds ne sont pas faciles à gérer pour la partie contiguë. De plus, il est difficile de donner un maximum pour la déclaration de cette partie si on peut ajouter des éléments. Sur l'exemple, Julie a un successeur en 1 (soit Jonatan) et un autre en 5 (soit Gontran). Chaque nœud a une liste (vide pour les feuilles) des successeurs de ce nœud.

3.1.3.b Allocation dynamique

Les éléments sont alloués au cours de la construction de l'arbre et reliés entre eux. Le nombre de pointeurs présents dans chaque nœud dépend du degré de l'arbre. Si le degré de chaque nœud est constant, cette mémorisation est parfaite. Sur l'exemple de l'arbre généalogique (voir Figure 55), le nombre maximum N d'enfants pour une personne est difficile à définir. De plus, il conduit à une perte importante de place puisqu'il faut prévoir N pointeurs pour chaque nœud, y compris les feuilles.

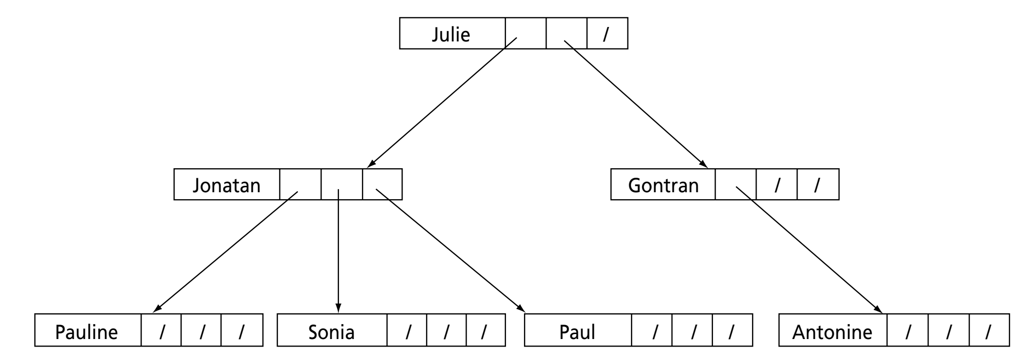


Figure 55 Allocation dynamique avec 3 descendants maximum.

La déclaration pourrait être la suivante :

```
typedef struct noeud* PNoeud;
typedef struct noeud {
  char nom [16];
  PNoeud p1;
  PNoeud p2,
  PNoeud p3;
} Noeud;
```

3.1.3.c Allocation contiguë (tableau ou fichier)

	Nom	p1	p2	p3
0	Julie	1	5	/
1	Jonatan	2	3	4
2	Pauline	/	/	/
3	Sonia	/	/	/
4	Paul	/	/	/
5	Gontran	6	/	/
6	Antonine	/	/	/

Figure 56 Allocation contiguë avec 3 descendants maximum.

L'arbre peut être mémorisé dans un espace contigu (voir Figure 56). L'espace mémoire est réservé avant le début de l'exécution en précisant un maximum. Cet espace peut être réservé en mémoire centrale ou sur mémoire secondaire. Les pointeurs sont alors des indices du tableau, ou des numéros d'enregistrements s'il s'agit d'un fichier. Si l'entrée 0 est utilisée, on peut choisir –1 pour indiquer l'absence de successeurs notée / sur le schéma.

D'où les déclarations :

```
#define NULLE    -1
#define MAXPERS   7

typedef int PNoeud;
typedef struct {
  char    nom [16];
  PNoeud  p1, p2, p3;
} Noeud

Noeud genealogique [MAXPERS];  // pour un tableau
```

Dans les cas où il y a des ajouts et des retraits de nœuds, on pourrait envisager une gestion en liste libre de l'espace non utilisé (voir § 2.9.1.c, page 87). Si un nœud a 10 successeurs, il faut envisager 10 pointeurs pour chaque nœud de l'arbre, d'où une perte de place. Notation : genealogique[1].nom représente *Jonatan* sur l'exemple.

3.2 LES ARBRES BINAIRES

3.2.1 Définition d'un arbre binaire

Un arbre binaire est un arbre ordonné tel que chaque nœud a au plus deux fils et quand il n'y en a qu'un, on distingue le fils droit du fils gauche.

3.2.2 Transformation d'un arbre n-aire en un arbre binaire

Lorsque le nombre de successeurs des nœuds est variable, il est souvent préférable de convertir l'arbre n-aire en un arbre binaire équivalent. On mémorise alors pour chaque nœud, un pointeur vers son premier fils et un pointeur vers son frère immédiatement plus jeune. Les liens *premier fils* et *frère immédiatement plus jeune* suffisent pour représenter tout arbre n-aire. Sur la Figure 57, le trait vertical représente le premier fils ; le trait horizontal représente le frère immédiatement plus jeune.

Exemple :

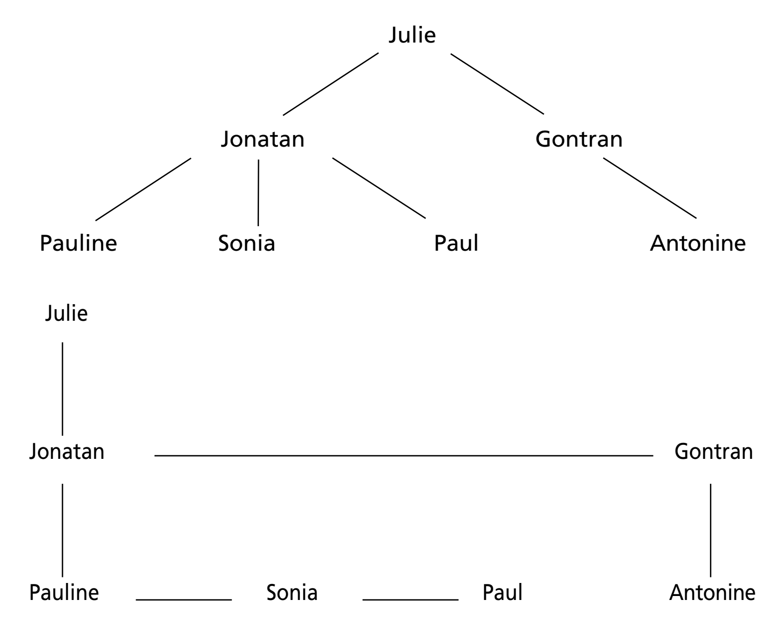


Figure 57 Transformation d'un arbre n-aire en un arbre binaire.

3.2.3 Mémorisation d'un arbre binaire

3.2.3.a Arbre généalogique

Allocation dynamique : création d'un nœud au fur et à mesure des besoins.

La mémorisation de l'arbre en allocation entièrement dynamique est donnée sur le schéma de la Figure 58. Chaque nœud a au plus deux successeurs.

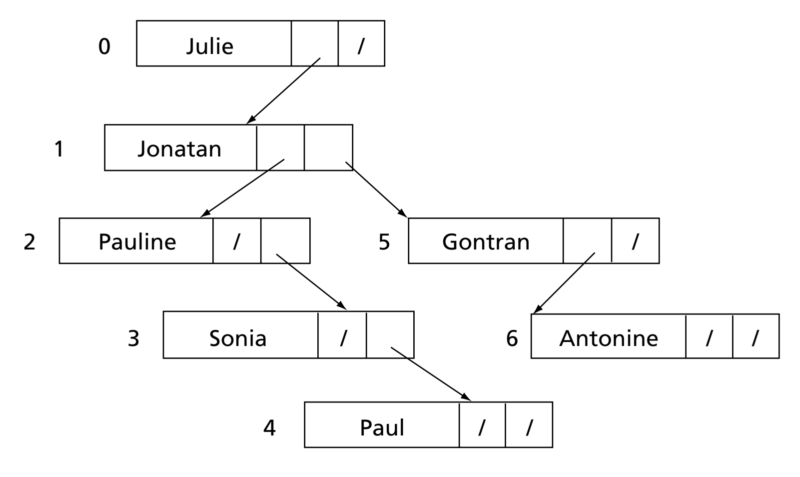


Figure 58 Arbre binaire avec une représentation à 45° des liens fils et frère.

Les fonctions suivantes permettent de créer de nouveaux nœuds et de construire l'arbre généalogique de l'exemple. Comme pour les listes, afin de donner plus de généralités au module, les informations spécifiques de l'application sont repérées par un pointeur nommé *reference* qui pointe sur l'objet de l'application (un entier, une personne, etc.). Le pointeur est de type indifférencié soit de type void* ou Objet* avec les notations suivantes.

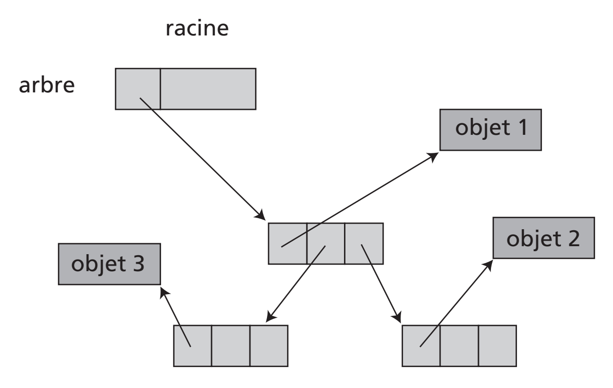


Figure 59 Dessin d'un arbre composé de nœuds. Chaque nœud référence un objet.

```
typedef void Objet;

typedef struct noeud {
  Objet*        reference;
  struct noeud* gauche;
  struct noeud* droite;
  int    factEq;    // facteur d'équilibre : si l'arbre est équilibré
} Noeud;

typedef struct {
  Noeud* racine;
  char* (*toString) (Objet*);
  int   (*comparer) (Objet*, Objet*);
} Arbre;
```

Les fonctions suivantes fournissent ou changent la valeur d'un paramètre d'une structure de type Arbre (accesseurs).

```
Noeud* getracine              (Arbre* arbre);
Objet* getobjet               (Noeud* racine);
void   setracine              (Arbre* arbre, Noeud* racine);
void   settoString            (Arbre* arbre, char* (*toString) (Objet*));
void   setcomparer            (Arbre* arbre,
                               int (*comparer) (Objet*, Objet*));
```

Ces fonctions créent un nœud ou une feuille (une structure Nœud) :

```
// création d'un noeud interne contenant objet,
// gauche comme pointeur de SAG, et droite comme pointeur de SAD
Noeud* cNd (Objet* objet, Noeud* gauche, Noeud* droite) {
  Noeud* nouveau     = new Noeud();
```

```
  nouveau->reference = objet;
  nouveau->gauche    = gauche;
  nouveau->droite    = droite;
  return nouveau;
}

// création d'un noeud feuille contenant objet
Noeud* cNd (Objet* objet) {
  return cNd (objet, NULL, NULL);
}

// création d'une feuille contenant objet
Noeud* cF (Objet* objet) {
  return cNd (objet, NULL, NULL);
}
```

Ces fonctions créent ou initialisent un arbre (une structure Arbre).

```
void initArbre (Arbre* arbre, Noeud* racine,
          char* (*toString) (Objet*), int (*comparer) (Objet*, Objet*)) {
  arbre->racine   = racine;
  arbre->toString = toString;
  arbre->comparer = comparer;
}

Arbre* creerArbre (Noeud* racine, char* (*toString) (Objet*),
                                     int (*comparer) (Objet*, Objet*)) {
  Arbre* arbre = new Arbre();
  initArbre (arbre, racine, toString, comparer);
  return arbre;
}

Arbre* creerArbre (Noeud* racine) {
  return creerArbre (racine, toChar, comparerCar);
}

Arbre* creerArbre () {
  return creerArbre (NULL, toChar, comparerCar);  // valeurs par défaut
}
```

Les fonctions suivantes créent l'arbre généalogique particulier de la Figure 58.

```
Noeud* cF (char* message) {
  return cF ( (Objet*) message);
}

Noeud* cNd (char* message, Noeud* gauche, Noeud* droite) {
  return cNd ( (Objet*) message, gauche, droite);
}

// créer un arbre binaire généalogique
Arbre* creerArbreGene () {
  Noeud* racine =
    cNd ( "Julie",
          cNd ( "Jonatan",
                cNd ( "Pauline",
                      NULL,
                      cNd ( "Sonia", NULL, cF ("Paul") )
                    ),
```

```
                cNd ( "Gontran", cF ("Antonine"), NULL)
               ),
          NULL
        );
  return creerArbre (racine);
}
```

Allocation contiguë

L'allocation peut aussi se faire dans un tableau en mémoire centrale, ou dans un fichier en accès direct si le volume des données est important, ou si l'arbre doit être conservé d'une session à l'autre.

	nom	gauche	droite
0	Julie	1	/
1	Jonatan	2	5
2	Pauline	/	3
3	Sonia	/	4
4	Paul	/	/
5	Gontran	6	/
6	Antonine	/	/

Figure 60 Arbre binaire en allocation contiguë (tableau ou fichier).

Le schéma de la Figure 60 peut représenter soit :

- un tableau en mémoire centrale
- un fichier en accès direct en mémoire secondaire (disque).

3.2.3.b Expression arithmétique

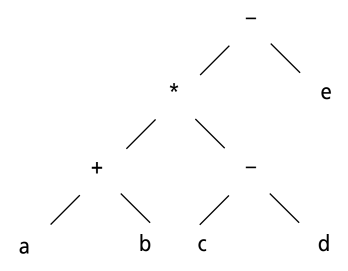
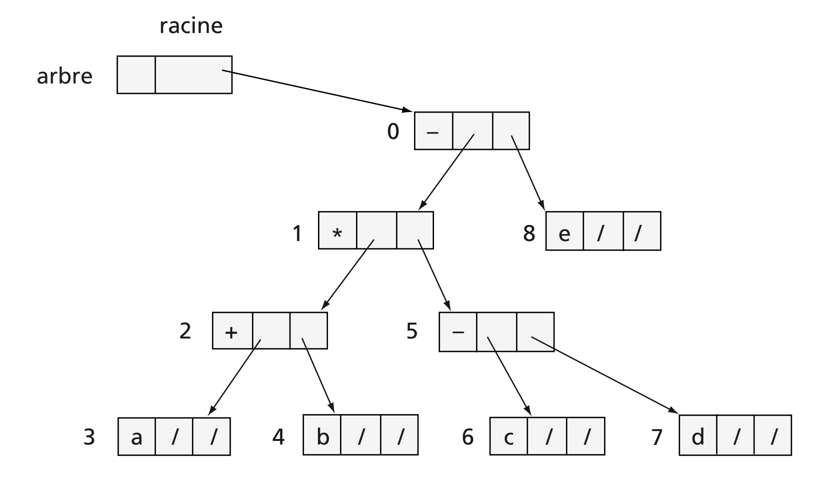


Figure 61 Arbre binaire de ((a+b)*(c-d))-e.

L'arbre de la Figure 61 peut être créé comme suit à l'aide des fonctions `cF()` et `cNd()` vues précédemment.

```
// créer un arbre binaire (expression arithmétique)
Arbre* creerArbreExp () {
  Noeud* racine =
    cNd ( "-",
          cNd ( "*",
                 cNd ("+", cF ("a"), cF ("b") ),
                 cNd ("-", cF ("c"), cF ("d") )
               ),
          cF  ("e")
        );
  return creerArbre (racine);
}
```

Mémorisation dynamique puis contiguë de l'expression arithmétique



	nom	gauche	droite
0	–	1	8
1	*	2	5
2	+	3	4
3	a	/	/
4	b	/	/
5	–	6	7
6	c	/	/
7	d	/	/
8	e	/	/

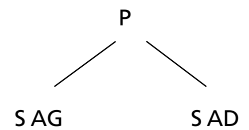
Figure 62 Mémorisation de l'arbre binaire de ((a+b)*(c-d))-e.

3.2.4 Parcours d'un arbre binaire

Un algorithme de parcours d'arbre est un procédé permettant d'accéder à chaque nœud de l'arbre. Un certain traitement est effectué pour chaque nœud (test, écriture, comptage, etc.), mais le parcours est indépendant de cette action et commun à des algorithmes qui peuvent effectuer des traitements très divers comme rechercher les enfants de Gontran, compter la descendance de Julie, compter le nombre de garçons, ajouter un fils à Paul, etc. On distingue deux catégories de parcours d'arbres : les parcours en profondeur et les parcours en largeur. Dans le parcours en profondeur, on explore branche par branche alors que dans le parcours en largeur on explore niveau par niveau.

3.2.4.a Les différentes méthodes de parcours en profondeur d'un arbre

Il y a 6 types de parcours possibles (P : père, SAG : sous-arbre gauche, SAD : sous-arbre droit). Nous ne considérons dans la suite de ce chapitre que les parcours gauche-droite. Les parcours droite-gauche s'en déduisent facilement par symétrie.



Ces parcours sont appelés parcours en **profondeur** car on explore une branche de l'arbre le plus profond possible avant de revenir en arrière pour essayer un autre chemin.

	gauche - droite	droite - gauche
préfixé	P . SAG . SAD	P . SAD . SAG
infixé	SAG . P . SAD	SAD . P . SAG
postfixé	SAG . SAD . P	SAD . SAG . P

Figure 63 Les 6 types de parcours d'un arbre binaire.

Dans un parcours d'arbre gauche-droite, un nœud est visité trois fois :

- lors de la première rencontre du nœud, avant de parcourir le sous-arbre gauche.
- après parcours du sous-arbre gauche, avant de parcourir le sous-arbre droit.
- après examens des sous-arbres gauche et droit.

L'action à effectuer sur le nœud peut se faire lors de la visite (a), (b) ou (c).

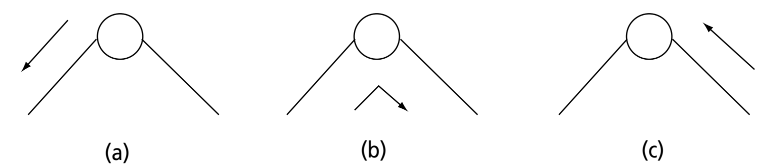


Figure 64 Les 3 visites d'un nœud lors d'un parcours d'arbre binaire.

3.2.4.b Parcours sur l'arbre binaire de l'arbre généalogique

Parcours préfixé

Le premier type de parcours est appelé parcours préfixé. Il faut traiter le nœud lors de la **première** visite, puis explorer le sous-arbre gauche (en appliquant la même méthode) avant d'explorer le sous-arbre droit. Sur la Figure 58, *Jonatan* est traité avant son SAG (*Pauline Sonia Paul*) et avant son SAD (*Gontran Antonine*). La procédure se schématise comme suit :

- traitement de la racine
- traitement du sous-arbre gauche
- traitement du sous-arbre droit

Sur l'exemple, cela conduit au parcours de la Figure 65. Pour chaque nœud, on trouve le nom du nœud concerné, les éléments du SAG, puis les éléments du SAD.

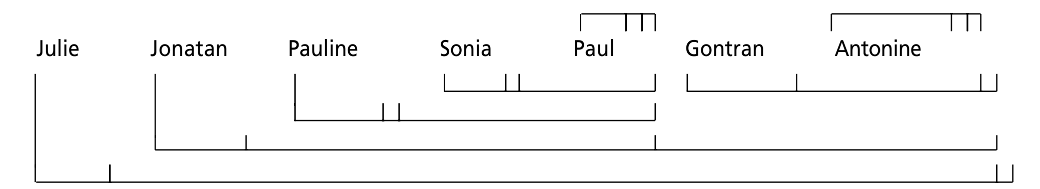


Figure 65 Parcours préfixé de l'arbre généalogique de la Figure 58.

Parcours infixé

Dans un parcours infixé, le nœud est traité lors de la **deuxième** visite, après avoir traité le sous-arbre gauche, mais avant de traiter le sous-arbre droit. La Figure 66 indique l'ordre de traitement des nœuds de l'arbre généalogique. Un nœud se trouve entre son SAG et son SAD. *Jonatan* par exemple se trouve entre son SAG (*Pauline Sonia Paul*) et son SAD (*Antonine Gontran*). La procédure se schématise comme suit :

- traitement du sous-arbre gauche
- traitement de la racine
- traitement du sous-arbre droit

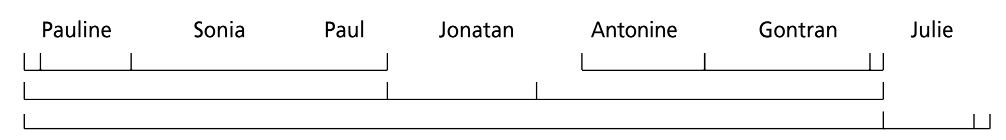


Figure 66 Parcours infixé de l'arbre généalogique de la Figure 58.

Parcours postfixé

En parcours postfixé, le nœud est traité lors de la **troisième** visite, après avoir traité le SAG et le SAD. La Figure 67 indique par exemple que *Jonatan* est traité après son

SAG (*Paul Sonia Pauline*) et après son SAD (*Antonine Gontran*). La procédure à suivre est donnée ci-dessous :

- traitement du sous-arbre gauche
- traitement du sous-arbre droit
- traitement de la racine

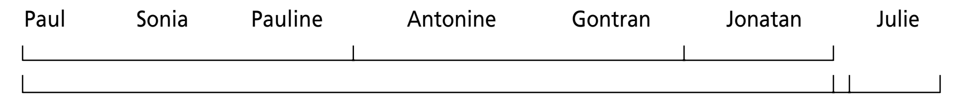


Figure 67 Parcours postfixé de l'arbre généalogique de la Figure 58.

Exercice 15 - Parcours d'arbres droite-gauche

Donner sur l'exemple de la Figure 58, les parcours préfixé, infixé, postfixé en parcours droite-gauche (voir Figure 63).

3.2.4.c Parcours sur l'arbre binaire de l'expression arithmétique

Sur l'arbre binaire de l'expression arithmétique, les parcours correspondent à une écriture préfixée, infixée ou postfixée de cette expression.

Parcours préfixé

L'opérateur est traité avant ses opérandes

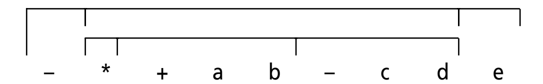


Figure 68 Parcours préfixé de l'expression arithmétique de la Figure 62.

Parcours infixé

L'opérateur se trouve entre ses deux opérandes.

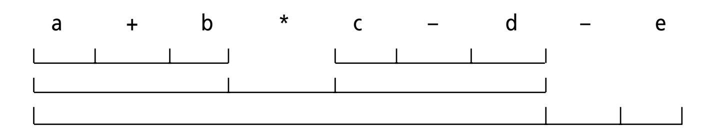


Figure 69 Parcours infixé de l'expression arithmétique de la Figure 62.

L'expression infixée est ambiguë et peut être interprétée comme :

a + (b * c) - d - e ou (a + b) * (c - d) - e

Il faut utiliser des parenthèses pour lever l'ambiguïté. C'est la notation habituelle d'une expression arithmétique dans les langages de programmation. En l'absence de parenthèses, des priorités entre opérateurs permettent aux compilateurs de choisir une des interprétations possibles.

Parcours postfixé
L'opérateur se trouve après ses opérandes.

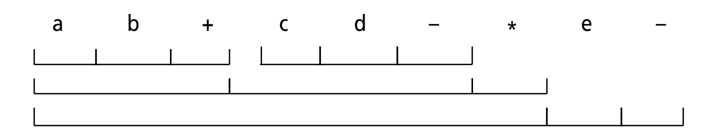


Figure 70 Parcours postfixé de l'expression arithmétique de la Figure 62.

3.2.4.d Les algorithmes de parcours d'arbre binaire

Les fonctions de parcours découlent directement des algorithmes vus sur les exemples précédents. Le simple changement de la place de l'ordre d'écriture conduit à un traitement préfixé, infixé ou postfixé. La fonction *toString()*, passée en paramètre de *prefixe()* fournit une chaîne de caractères spécifiques de l'objet traité. Cette chaîne est imprimée lors du printf. La fonction *toString()* est dépendante de l'application et passée en paramètre lors de la création de l'arbre. Par défaut, les objets référencés dans chaque nœud sont des chaînes de caractères.

Algorithme de parcours préfixé

```
// toString fournit la chaîne de caractères à écrire pour un objet donné
static void prefixe (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    printf ("%s ", toString (racine->reference));
    prefixe (racine->gauche, toString);
    prefixe (racine->droite, toString);
  }
}

// parcours préfixé de l'arbre
void prefixe (Arbre* arbre) {
  prefixe (arbre->racine, arbre->toString);
}
```

Le déroulement de l'algorithme récursif *prefixe()* sur la Figure 62 où les pointeurs ont été remplacés pour l'explication par des adresses de 0 à 8 est schématisé sur la Figure 71. Les adresses des nœuds en allocation dynamique sont normalement quelconques et dispersées en mémoire. L'appel *prefixe()* avec le nœud racine 0 entraîne un appel récursif qui consiste à traiter le SAG, d'où un appel à *prefixe()* avec pour nouvelle racine 1 qui à son tour déclenche une cascade d'appels récursifs. Plus tard, on fera un appel à *prefixe()* avec un pointeur sur SAD en 8. Le déroulement de l'exécution de l'algorithme est schématisé ligne par ligne, de haut en bas, et de gauche à droite.

prefixe (0);	racine = 0; printf (-); prefixe (1);	racine = 1; printf (*); prefixe (2);	racine = 2; printf (+); prefixe (3);	racine = 3; printf (a); prefixe (NULL); prefixe (NULL);
			prefixe (4);	racine = 4; printf (b); prefixe (NULL); prefixe (NULL);
		prefixe (5);	racine = 5; printf (-); prefixe (6);	Racine = 6; printf (c); prefixe (NULL); prefixe (NULL);
			prefixe (7);	racine = 7; printf (d); prefixe (NULL); prefixe (NULL);
	prefixe (8);	racine = 8; printf (e); prefixe (NULL); prefixe (NULL);		

Figure 71 Parcours préfixé de l'arbre binaire de la Figure 62.

Algorithme de parcours infixé

Le nœud racine est traité (écrit) entre les deux appels récursifs.

```
// toString fournit la chaîne de caractères à écrire pour un objet
static void infixe (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    infixe (racine->gauche, toString);
    printf ("%s ", toString (racine->reference));
    infixe (racine->droite, toString);
  }
}

// parcours infixé de l'arbre
void infixe (Arbre* arbre) {
  infixe (arbre->racine, arbre->toString);
}
```

Algorithme de parcours postfixé

Le nœud racine est traité après les deux appels récursifs.

```
// toString fournit la chaîne de caractères à écrire pour un objet
static void postfixe (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    postfixe (racine->gauche, toString);
    postfixe (racine->droite, toString);
    printf ("%s ", toString (racine->reference));
  }
}

// parcours postfixé de l'arbre
void postfixe (Arbre* arbre) {
  postfixe (arbre->racine, arbre->toString);
}
```

Parcours préfixé avec indentation

Les écritures concernant les objets des nœuds visités sont décalées (indentées) pour mieux mettre en évidence la structure de l'arbre binaire. Le niveau est augmenté de 1 à chaque fois que l'on descend à gauche ou à droite dans l'arbre binaire.

```
// toString fournit la chaîne de caractères à écrire pour un objet
// niveau indique l'indentation à faire
static void indentationPrefixee (Noeud* racine,
                                   char* (*toString) (Objet*), int niveau) {
  if (racine != NULL) {
    printf ("\n");
    for (int i=1; i<niveau; i++) printf ("%5s", " ");
    printf ("%s ", toString (racine->reference));
    indentationPrefixee (racine->gauche, toString, niveau+1);
    indentationPrefixee (racine->droite, toString, niveau+1);
  }
}

void indentationPrefixee (Arbre* arbre) {
  indentationPrefixee (arbre->racine, arbre->toString, 1);
}
```

Résultats du parcours préfixé avec indentation :

L'exécution de la fonction *indentationPrefixee()* sur l'exemple de la Figure 62 conduit aux résultats suivants où la structure de l'arbre binaire est mise en évidence.

```
-
     *
          +
               a
               b
          -
               c
               d
     e
```

3.2.4.e Recherche d'un nœud de l'arbre

La fonction *trouverNoeud()* recherche récursivement le nœud contenant les informations définies dans objet, dans l'arbre commençant en racine. C'est un *parcours d'arbre interrompu* (on s'arrête quand on a trouvé un pointeur sur l'objet concerné). Si l'objet n'est pas dans l'arbre, la fonction retourne NULL.

Algorithme : la comparaison de deux objets est définie par la fonction *comparer()* passée en paramètre et spécifique des objets traités dans l'application. Cette fonction est définie lors de la création de l'arbre. Par défaut, il s'agit d'un arbre de chaînes de caractères.

La recherche de objet dans un arbre vide retourne la valeur NULL (pas trouvé). Si le nœud pointé par racine contient l'objet que l'on cherche alors le résultat est le

pointeur racine ; sinon, on cherche objet dans le SAG ; si objet n'est pas dans le SAG, on le cherche dans le SAD.

```
static Noeud* trouverNoeud (Noeud* racine, Objet* objet,
                                      int (*comparer) (Objet*, Objet*)) {
  Noeud* pNom;
  if (racine == NULL) {
    pNom = NULL;
  } else if (comparer (racine->reference, objet) == 0) {
    pNom = racine;
  } else {
    pNom = trouverNoeud (racine->gauche, objet, comparer);
    if (pNom == NULL) pNom = trouverNoeud (racine->droite, objet,
                                                          comparer);
  }
  return pNom;
}

// recherche le noeud objet dans l'arbre
Noeud* trouverNoeud (Arbre* arbre, Objet* objet) {
  return trouverNoeud (arbre->racine, objet, arbre->comparer);
}
```

Cette procédure est utile pour retrouver un nœud particulier de l'arbre et déclencher un traitement à partir de ce nœud. On peut par exemple appeler *trouverNœud()* pour obtenir un pointeur sur le nœud *Jonatan* de la Figure 58 et effectuer une énumération indentée à partir de ce nœud en appelant la fonction *indentationPrefixee()*.

3.2.4.f Parcours en largeur dans un arbre

Une autre méthode de parcours des arbres consiste à les visiter étage par étage, comme si on faisait une coupe par niveau. Ainsi, sur l'arbre généalogique binaire de la Figure 58, le parcours en largeur est le suivant : Julie/Jonatan/Pauline-Gontran/Sonia-Antonine/Paul. Sur l'arbre binaire de l'expression arithmétique de la Figure 61, le parcours en largeur est : - * e + - a b c d. Ce parcours nécessite l'utilisation d'une file d'attente contenant initialement la racine. On extrait l'élément en tête de la file, et on le remplace par ses successeurs à gauche et à droite jusqu'à ce que la file soit vide. Dans les parcours d'arbres, on effectue plutôt des parcours en profondeur, bénéficiant ainsi des mécanismes automatiques de retour en arrière de la récursivité. Le parcours en largeur est effectué lorsque les résultats l'imposent comme pour le dessin d'un arbre binaire (voir § 3.2.8, page 127).

La fonction **enLargeur**() effectue un parcours en largeur des nœuds de l'arbre.

Exemple de résultats pour l'arbre généalogique :

```
Parcours en largeur
Julie Jonatan Pauline Gontran Sonia Antonine Paul
```

```
static void enLargeur (Noeud* racine, char* (*toString) (Objet*)) {
  Liste* li = creerListe();
  insererEnFinDeListe (li, racine);

  while (!listeVide (li) ) {
    Noeud* extrait = (Noeud*) extraireEnTeteDeListe (li);
    printf ("%s ", toString (extrait->reference));
    if (extrait->gauche != NULL) insererEnFinDeListe (li,
                                                       extrait->gauche);
    if (extrait->droite != NULL) insererEnFinDeListe (li,
                                                       extrait->droite);
  }
}

// parcours en largeur de l'arbre
void enLargeur (Arbre* arbre) {
  enLargeur (arbre->racine, arbre->toString);
}
```

La fonction **EnLargeurParEtape**() effectue un parcours en largeur des nœuds de l'arbre en effectuant un traitement en fin de chaque étage (aller à la ligne ici). Il faut utiliser 2 listes : une liste contenant les pointeurs sur les nœuds de l'étage courant, et une autre contenant les successeurs des nœuds courants qui deviendront étage courant à l'étape suivante.

Exemple de résultats :

```
Parcours en largeur par étage
Julie
Jonatan
Pauline Gontran
Sonia Antonine
Paul
```

```
static void enLargeurParEtage (Noeud* racine, char* (*toString) (Objet*)) {
  Liste* lc = creerListe();   // liste courante
  Liste* ls = creerListe();   // liste suivante
  insererEnFinDeListe (lc, racine);

  while (!listeVide (lc)) {
    while (!listeVide (lc) ) {
      Noeud* extrait = (Noeud*) extraireEnTeteDeListe (lc);
      printf ("%s ", toString (extrait->reference));
      if (extrait->gauche != NULL) insererEnFinDeListe (ls,
                                                         extrait->gauche);
      if (extrait->droite != NULL) insererEnFinDeListe (ls,
                                                         extrait->droite);
    }

    printf ("\n");             // fin d'un étage
    recopierListe (lc, ls);    // ls vide
  }
}

void enLargeurParEtage (Arbre* arbre) {
  enLargeurParEtage (arbre->racine, arbre->toString);
}
```

3.2.5 Propriétés de l'arbre binaire

3.2.5.a Taille d'un arbre binaire

La taille d'un arbre est le nombre de nœuds de cet arbre. La taille à partir du nœud pointé par racine est de 0 si l'arbre est NULL, et vaut 1 (le nœud pointé par racine) plus le nombre de nœuds du sous-arbre gauche et plus le nombre de nœuds du sous-arbre droit sinon.

```
int taille (Noeud* racine) {
  if (racine == NULL) {
    return 0;
  } else {
    return 1 + taille (racine->gauche) + taille (racine->droite);
  }
}

// nombre de noeuds de l'arbre
int taille (Arbre* arbre) {
  return taille (arbre->racine);
}
```

Dans le cas où racine pointe sur une feuille par exemple Paul sur la Figure 58,

```
taille (racine) = 1 + taille (NULL) + taille (NULL)
                = 1 + 0             + 0
                = 1
```

3.2.5.b Feuilles de l'arbre binaire

La fonction *estFeuille()* est une fonction booléenne qui indique si le nœud pointé par racine est une feuille (n'a pas de successeur).

```
// Le noeud racine est-il une feuille ?
booleen estFeuille (Noeud* racine) {
  return (racine->gauche==NULL) && (racine->droite==NULL);
}
```

La fonction *nbFeuilles()* compte le nombre de feuilles de l'arbre **binaire** à partir du nœud racine. Si l'arbre est vide, le nombre de feuilles est 0 ; sinon si racine repère une feuille, le nombre de feuilles est de 1, sinon, le nombre de feuilles en partant du nœud racine est le nombre de feuilles du SAG, plus le nombre de feuilles du SAD. Sur l'arbre binaire de la Figure 58, le nombre de feuilles de l'arbre binaire est de 2 (*Paul* et *Antonine*), alors que le nombre de feuilles de l'arbre n-aire est de 4 sur la Figure 57.

```
static int nbFeuilles (Noeud* racine) {
  if (racine == NULL) {
    return 0;
  } else if ( estFeuille (racine) ) {
    return 1;
  } else {
    return nbFeuilles (racine->gauche) + nbFeuilles (racine->droite);
  }
}
```

```
// fournir le nombre de feuilles de l'arbre binaire
int nbFeuilles (Arbre* arbre) {
  return nbFeuilles (arbre->racine);
}
```

La fonction *listerFeuilles()* énumère les feuilles de l'arbre binaire. C'est un parcours préfixé d'arbre avec écriture lorsque racine pointe sur une feuille. Sur l'arbre binaire de la Figure 58, la fonction liste les deux feuilles de l'arbre binaire : *Paul* et *Antonine*.

```
static void listerFeuilles (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    if (estFeuille (racine)) {
      printf ("%s ", toString (racine->reference));
    } else {
      listerFeuilles (racine->gauche, toString);
      listerFeuilles (racine->droite, toString);
    }
  }
}

// lister les feuilles de l'arbre binaire
void listerFeuilles (Arbre* arbre) {
  listerFeuilles (arbre->racine, arbre->toString);
}
```

3.2.5.c Valeur du plus long identificateur des nœuds de l'arbre

Cette fonction fournit la longueur du plus long des identificateurs de l'arbre. Si l'arbre est vide, la longueur maximum est 0 ; sinon le plus long identificateur en partant du nœud racine est le maximum des 3 valeurs suivantes : le plus long du sous-arbre gauche (SAG), le plus long du sous-arbre droit (SAD), et la longueur de l'identificateur de l'objet du nœud racine. Sur l'arbre binaire de la Figure 58, la longueur du plus long identificateur est 8 (*Antonine)*.

```
static int maxIdent (Noeud* racine, char* (*toString) (Objet*) ) {
  int lg;      // longueur max
  if (racine == NULL) {
    lg = 0;
  } else {
    lg = max ( maxIdent (racine->gauche, toString),
               maxIdent (racine->droite, toString) );
    lg = max (lg, strlen(toString(racine->reference)));
  }
  return lg;
}

// longueur du plus long identificateur de l'arbre
int maxIdent (Arbre* arbre) {
  return maxIdent (arbre->racine, arbre->toString);
}
```

3.2.5.d Somme des longueurs des identificateurs des nœuds de l'arbre

Il s'agit de parcourir l'arbre et de faire la somme des longueurs des identificateurs des objets de l'arbre. Cette fonction est utilisée ultérieurement pour effectuer le dessin de l'arbre. Si l'arbre est vide, la somme des longueurs est nulle, sinon, pour l'arbre commençant en racine, la somme des longueurs des identificateurs est la somme des longueurs du SAG, plus la somme des longueurs du SAD, plus la longueur de l'objet du nœud racine.

```
static int somLgIdent (Noeud* racine, char* (*toString) (Objet*)) {
  int s;
  if (racine == NULL) {
    s = 0;
  } else {
    s = somLgIdent (racine->gauche, toString) +
        somLgIdent (racine->droite, toString) +
        strlen(toString(racine->reference));
  }
  return s;
}

// somme des longueurs des identificateurs de l'arbre
int somLgIdent (Arbre* arbre) {
  return somLgIdent (arbre->racine, arbre->toString);
}
```

3.2.5.e Hauteur d'un arbre binaire, arbre binaire dégénéré

La hauteur d'un arbre binaire pointé par racine est le nombre de nœuds entre racine (compris) et la feuille de la plus longue branche partant de racine. Si l'arbre est vide, la hauteur est nulle. Sinon la hauteur en partant de racine est le maximum de la hauteur de SAG et de SAD auquel on ajoute 1 pour le nœud racine. La hauteur d'une feuille est de 1. Sur l'arbre binaire de la Figure 58, la hauteur du nœud *Jonatan* est de 4 (longueur de la plus longue branche en partant de *Jonatan* vers *Paul*).

```
static int hauteur (Noeud* racine) {
  if (racine == NULL) {
    return 0;
  } else {
    return 1 + max (hauteur (racine->gauche),
                    hauteur (racine->droite) );
  }
}

// hauteur de l'arbre
int hauteur (Arbre* arbre) {
  return hauteur (arbre->racine);
}
```

L'arbre binaire est dégénéré si pour chaque nœud interne, il n'y a qu'un successeur, le dernier nœud étant une feuille. La méthode consiste à parcourir l'arbre et à fournir faux quand on rencontre un nœud ayant deux successeurs. Il y a quatre cas. Si on atteint la feuille terminale, dégénéré est vrai. Si les deux pointeurs sont différents de NULL alors l'arbre n'est pas dégénéré. Si le SAG est vide, le résultat

dépend de l'examen de SAD ; et si SAD est vide, le résultat dépend de l'examen de SAG. Sur l'arbre binaire de la Figure 58, le sous-arbre partant du nœud *Pauline* est dégénéré.

```
// racine != NULL
static booleen degenere (Noeud* racine) {
  booleen d;
  if (estFeuille(racine)) {
    d = vrai;
  } else if ( (racine->gauche != NULL) && (racine->droite != NULL) ) {
    d = faux;
  } else if (racine->gauche==NULL) {
    d = degenere (racine->droite);
  } else {
    d = degenere (racine->gauche);
  }
  return d;
}

// l'arbre est-il dégénéré ?
int degenere (Arbre* arbre) {
  return degenere (arbre->racine);
}
```

3.2.6 Duplication, destruction d'un arbre binaire

La fonction **dupliquerArbre**() crée une copie de l'arbre passé en paramètre et fournit un pointeur sur la racine du nouvel arbre créé. Si l'arbre à dupliquer est vide, sa copie est vide (NULL). Sinon, il faut créer un nouveau nœud *nouveau* qui référence le même objet que racine (copie du pointeur de l'objet, pas des zones pointées).

Il faut dupliquer le SAG ce qui fournit un pointeur sur ce nouveau SAG qui est rangé dans le champ gauche de *nouveau*, et de même, il faut dupliquer le SAD et ranger la racine de ce nouveau SAD dans le champ droit de *nouveau*. La fonction retourne un pointeur sur le nœud créé.

```
// dupliquer l'arbre racine,
// sans dupliquer les objets de l'arbre
static Noeud* dupliquerArbre (Noeud* racine) {
  if (racine==NULL) {
    return NULL;
  } else {
    Noeud* nouveau  = cNd (racine->reference);
    nouveau->gauche = dupliquerArbre (racine->gauche);
    nouveau->droite = dupliquerArbre (racine->droite);
    return nouveau;
  }
}

Arbre* dupliquerArbre (Arbre* arbre) {
  Noeud* nrac = dupliquerArbre (arbre->racine);
  return creerArbre (nrac, arbre->toString, arbre->comparer);
}
```

Si on veut dupliquer les objets référencés dans chaque nœud de l'arbre, il faut passer en paramètre de dupliquer une fonction capable d'effectuer une copie de l'objet. Cette fonction dépend de l'application.

```
// dupliquer l'arbre racine,
// en dupliquant les objets de l'arbre
// la fonction cloner permet de dupliquer l'objet du noeud
static Noeud* dupliquerArbre (Noeud* racine, Objet* (*cloner) (Objet*)) {
  if (racine==NULL) {
    return NULL;
  } else {
    Noeud* nouveau  = cNd (cloner (racine->reference));
    nouveau->gauche = dupliquerArbre (racine->gauche, cloner);
    nouveau->droite = dupliquerArbre (racine->droite, cloner);
    return nouveau;
  }
}

// cloner un arbre
Arbre* dupliquerArbre (Arbre* arbre, Objet* (*cloner) (Objet*)) {
  Noeud* nrac = dupliquerArbre (arbre->racine, cloner);
  return creerArbre (nrac, arbre->toString, arbre->comparer);
}
```

La fonction *detruireArbre()* effectue un parcours postfixé de l'arbre et détruit le nœud pointé par racine lors de la troisième visite du nœud (voir Figure 64, page 114) après destruction du SAG et destruction du SAD. En fin d'exécution, l'arbre est vide ; sa racine vaut NULL. Pour modifier la racine, il faut passer en paramètre l'adresse de la racine. Les objets ne sont pas détruits.

```
static void detruireArbre (Noeud** pracine) {
  Noeud* racine = *pracine;
  if (racine != NULL) {
    detruireArbre (&racine->gauche);
    detruireArbre (&racine->droite);
    free (racine);
    *pracine = NULL;
  }
}

// détruire l'arbre et mettre le pointeur de la racine à NULL
// sans détruire les objets pointés
void detruireArbre (Arbre* arbre) {
  detruireArbre (&arbre->racine);
}
```

Pour détruire les objets, il faudrait passer en paramètre de *detruireArbre()* une fonction détruisant l'objet et ses composantes (effectuant le contraire de *cloner()* vu ci-dessus).

```
static void detruireArbre (Noeud** pracine,
                                         void (*detruireObjet) (Objet*)) {
  Noeud* racine = *pracine;
  if (racine != NULL) {
    detruireArbre (&racine->gauche, detruireObjet);
    detruireArbre (&racine->droite, detruireObjet);
```

```
    detruireObjet (racine->reference);
    free (racine);
    *pracine = NULL;
  }
}

// détruire l'arbre et mettre le pointeur de la racine à NULL
// en détruisant les objets pointés
void detruireArbre (Arbre* arbre, void (*detruireObjet) (Objet*)) {
  detruireArbre (&arbre->racine, detruireObjet);
}
```

3.2.7 Égalité de deux arbres

La fonction ci-dessous teste l'égalité de deux arbres : les deux arbres doivent avoir la même structure et les mêmes informations.

```
// égalité de deux arbres
static booleen egaliteArbre (Noeud* racine1, Noeud* racine2,
                             int (*comparer) (Objet*, Objet*)) {
  booleen resu = faux;
  if ( (racine1==NULL) && (racine2==NULL) ) {
    resu = vrai;
  } else if ( (racine1!=NULL) && (racine2!=NULL) ) {
    if (comparer (racine1->reference, racine2->reference) == 0) {
      if (egaliteArbre (racine1->gauche, racine2->gauche, comparer) ) {
        resu = egaliteArbre (racine1->droite, racine2->droite, comparer);
      }
    }
  }
  return resu;
}

booleen egaliteArbre (Arbre* arbre1, Arbre* arbre2) {
  return egaliteArbre (arbre1->racine, arbre2->racine, arbre1->comparer);
}
```

3.2.8 Dessin d'un arbre binaire

La fonction *void dessinerArbre (Noeud* racine, FILE* fs) ;* dessine (en mode caractère) dans le fichier fs, l'arbre binaire pointé par racine. Si fs=stdout, le dessin se fait à l'écran.

La Figure 72 indique le résultat de l'exécution du programme de dessin sur l'arbre binaire de la Figure 58. Chaque identificateur occupe une colonne de la largeur de son identificateur. Les colonnes de début de chaque identificateur sont attribuées en faisant un parcours infixé. *Pauline*, le premier identificateur en parcours infixé (voir Figure 66), occupe les 7 premières colonnes de 0 à 6 ; sa position est centrée en colonne 3. *Sonia*, le deuxième identificateur occupe les 5 colonnes suivantes de 7 à 11 ; la position centrée de *Sonia* est donc 9. Ainsi *Paul* est en 14, *Jonatan* en 19, *Antonine* en 27, *Gontran* en 34 et *Julie* en 40. Ce calcul est effectué par la fonction `dupArb()` qui duplique l'arbre (voir § 3.2.6) en ajoutant la position de chaque identificateur dans chaque objet du nœud. Chaque objet

référencé par un nœud de l'arbre contient le message à écrire et le numéro de colonne de ce message (type NomPos, nom et position).

Exemple pour l'arbre généalogique

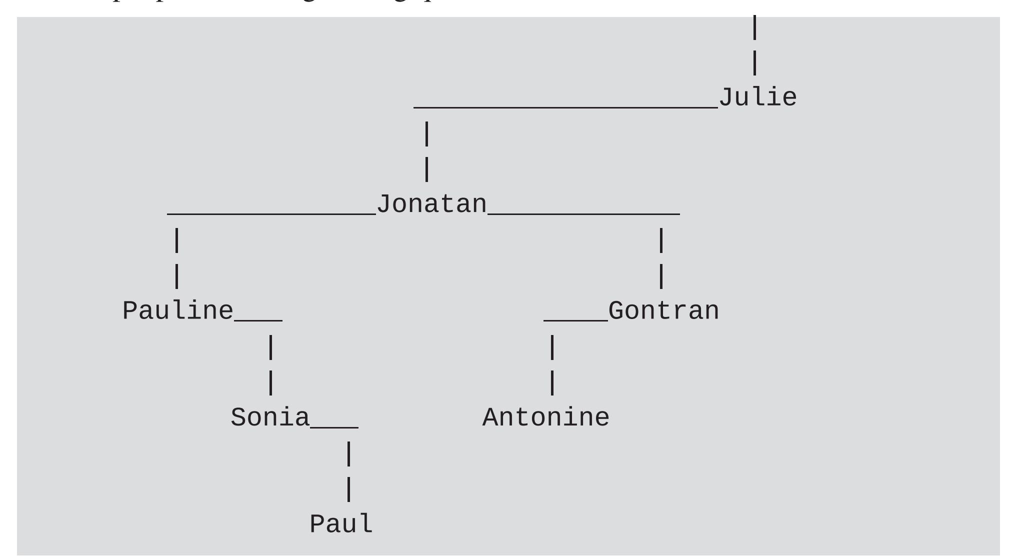


Figure 72 Dessin d'un arbre binaire.

```
// dessin de l'arbre

// message et position d'un noeud de l'arbre
typedef struct {
  char* message;   // message à afficher pour ce noeud
  int   position;  // position (n° de colonne) du noeud
} NomPos;


int  posNdC = 0;  // position du noeud courant : variable globale

// dupliquer l'arbre en remplaçant l'objet référencé par un objet NomPos
// contenant la chaîne de caractère à écrire et sa position.
static Noeud* dupArb (Noeud* racine, char* (*toString) (Objet*) ) {
  if (racine == NULL) {
    return NULL;
  } else {
    Noeud* nouveau  = new Noeud();
    NomPos* objet   = new NomPos();
    nouveau->reference = objet;
    objet->message  = toString (racine->reference);
    nouveau->gauche = dupArb (racine->gauche, toString);
    int lg          = strlen (toString(racine->reference));
    objet->position = posNdC + lg/2;
    posNdC         += lg;
    nouveau->droite = dupArb (racine->droite, toString);
    return nouveau;
  }
}
```

```
static Arbre* dupArb (Arbre* arbre) {
  posNdC = 0;      // globale pour dupArb
  Noeud* nrac = dupArb (arbre->racine, arbre->toString);
  return creerArbre (nrac, arbre->toString, NULL);
}
```

La fonction *dessinerArbre()* effectue un parcours en largeur de l'arbre binaire (voir § 3.2.4.f, page 120), c'est-à-dire qu'elle traite les nœuds de l'arbre binaire étage par étage : Julie/Jonatan/Pauline, Gontran/Sonia, Antonine/Paul. Pour cela, il faut utiliser deux listes. Une liste courante *lc* qui contient les pointeurs vers les nœuds de l'étage en cours de traitement (initialement un pointeur vers le nœud *Julie*, la racine de l'arbre). On parcourt une première fois cette liste pour écrire les barres verticales qui précèdent les noms des nœuds. On effectue un second parcours de la liste courante pour écrire les noms des nœuds, tracer les tirets en fonction de la position du SAG et du SAD et insérer dans la seconde liste *ls*, les nœuds à traiter lors de la prochaine étape (les SAG et SAD des nœuds de la liste courante). En fin de ce second parcours, on recopie la liste suivante dans la liste courante et on recommence le processus jusqu'à ce que la liste courante soit vide. On utilise le module *liste.h* pour la gestion des listes.

```
void dessinerArbre (Arbre* arbre, FILE* fs) {
  if (arbreVide(arbre)) {
    printf ("dessinerArbre  Arbre vide\n");
    return;
  }

  // largeur requise pour le dessin
  int lgFeuille = somLgIdent (arbre);
  char* ligne = (char*) malloc (lgFeuille+1);
  ligne [lgFeuille] = 0;

  // narbre : nouvel arbre dupliqué
  Arbre* narbre= dupArb (arbre);

  Liste* lc = creerListe();   // Liste des noeuds du même niveau
  insererEnFinDeListe (lc, narbre->racine);
  Liste* ls = creerListe();   // Liste des descendants de lc

  while (!listeVide (lc)) {
    // écrire les barres verticales des noeuds de la liste
    for (int i=0; i<lgFeuille; i++) ligne[i]=' ';
    ouvrirListe (lc);
    while (!finListe (lc) ) {
      Noeud* ptNd = (Noeud*) objetCourant (lc);
      NomPos* ptc = (NomPos*) ptNd->reference;
      ligne [ptc->position] = '|';
    }
    for (int i=1; i<=2; i++) fprintf (fs, "%s\n", ligne);

    // Pour chaque élément de la liste :
    //   écrire des tirets de la position du SAG à celle du SAD
    //   écrire le nom de l'élément à sa position
    for (int i=0; i<lgFeuille; i++) ligne[i]=' ';
```

```
    while (!listeVide (lc)) {
      Noeud* pNC  = (Noeud*) extraireEnTeteDeListe (lc);
      Noeud* pSAG = pNC->gauche;
      Noeud* pSAD = pNC->droite;
      char*  message  = ((NomPos*) pNC->reference)->message;
      int    lg       = strlen (message);
      int    position = ((NomPos*)pNC->reference)->position;
      int    posNom   = position - lg/2;
      int    posSAG   = pSAG==NULL ? position :
                                ((NomPos*)pSAG->reference)->position;
      int    posSAD   = pSAD==NULL ? position :
                                ((NomPos*)pSAD->reference)->position;
      if (pSAG != NULL) insererEnFinDeListe (ls, pSAG);
      if (pSAD != NULL) insererEnFinDeListe (ls, pSAD);

      for (int j=posSAG; j<=posSAD; j++) ligne [j] = '_';
      for (int j=0; j<lg; j++) ligne [posNom+j] = message[j];
    }

    fprintf (fs, "%s\n", ligne);
    recopierListe (lc, ls);  // ls vide
  }

  // détruire l'arbre intermédiaire
  detruireArbre (narbre);
}
```

3.2.9 Arbre binaire et questions de l'arbre n-aire

Dans de nombreuses applications, l'arbre est donné sous sa forme n-aire. C'est le cas de la nomenclature d'un objet comme par exemple les composantes et sous-composantes d'un avion, d'une voiture, d'une maison, de la Terre, ou du corps humain. Le nombre de sous-composantes étant variable (le degré des nœuds est variable), la mémorisation se fait par conversion de l'arbre n-aire en un arbre binaire (voir Figure 57). Il faut alors répondre à des questions de l'arbre n-aire en utilisant la mémorisation de l'arbre binaire. Ainsi, pour l'arbre généalogique de racine *Julie*, les feuilles de l'arbre n-aire correspondent aux personnes sans enfant (*Pauline*, *Sonia*, *Paul*, *Antonine*) et sont différentes des feuilles de l'arbre binaire (*Paul*, *Antonine*) qui n'ont pas d'intérêt pour l'application. Dans une nomenclature d'objet, les feuilles n-aires sont les composants (les pièces) de base qu'il faut assembler pour constituer l'objet.

> ***Remarque*** : ayant un pointeur sur un nœud de l'arbre binaire (fourni par exemple par la fonction *trouverNoeud()* (voir § 3.2.4.e, page 119), les descendants n-aires de ce nœud se trouvent dans le sous-arbre gauche. L'appel des diverses fonctions traitant de la descendance n-aire d'un nœud racine se fait donc en explorant le SAG du nœud de départ.

3.2.9.a Feuilles n-aires

Une feuille n-aire a un sous-arbre gauche vide, le pointeur sur le premier fils est à NULL. La fonction *listerFeuillesNAire()* est donc un parcours d'arbre binaire avec écriture pour les nœuds qui ont un sous-arbre gauche vide.

```
// lister les feuilles NAire à partir de racine
static void listerFeuillesNAire (Noeud* racine,
                                            char* (*toString) (Objet*)) {
  if (racine != NULL) {
    if (racine->gauche == NULL) printf ("%s ",
                                           toString (racine->reference));
    listerFeuillesNAire (racine->gauche, toString);
    listerFeuillesNAire (racine->droite, toString);
  }
}
```

Ayant un pointeur sur un nœud racine de l'arbre binaire, il faut explorer seulement le sous-arbre gauche de ce nœud racine. On crée un nouveau nœud racine ayant un sous-arbre droit vide.

```
void listerFeuillesNAire (Arbre* arbre) {
  if (arbre->racine != NULL) {
    Noeud* nrac   = cNd (arbre->racine->reference,
                                            arbre->racine->gauche, NULL);
    Arbre* narbre = creerArbre (nrac, arbre->toString, NULL);
    listerFeuillesNAire (narbre->racine, narbre->toString);
    free (nrac);
    free (narbre);
  }
}
```

Pour compter le nombre de feuilles n-aires, le principe est le même. Il faut compter au lieu d'écrire.

```
// fournir le nombre de feuilles n-aires à partir de racine
static int nbFeuillesNAire (Noeud* racine) {
  int n = 0;
  if (racine == NULL) {
    return 0;
  } else {
    if (racine->gauche == NULL) n = 1;
    return n + nbFeuillesNAire (racine->gauche)
             + nbFeuillesNAire (racine->droite);
  }
}

int nbFeuillesNAire (Arbre* arbre) {
  int n = 0;
  if (arbre->racine != NULL) {
    Noeud* nrac   = cNd (arbre->racine->reference,
                                            arbre->racine->gauche, NULL);
    Arbre* narbre = creerArbre (nrac, arbre->toString, NULL);
    n = nbFeuillesNAire (narbre->racine);
    free (nrac);
    free (narbre);
  }
  return n;
}
```

3.2.9.b Descendants n-aires

Cette fonction énumère tous les descendants n-aires d'un nœud. Si racine repère le nœud « Jonatan », descendantsNAire (racine, toString) fournit *Pauline Sonia Paul* (voir Figure 57). Il faut faire un parcours préfixé du SAG de *Jonatan*.

```
// fournir les descendants n-aires de racine
static void descendantsNAire (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    if (racine->gauche == NULL) {
      printf ("Pas de descendant pour %s\n",
                                              toString (racine->reference));
    } else {
      prefixe (racine->gauche, toString ); // descendants dans SAG
    }
  }
}

void descendantsNAire (Arbre* arbre) {
  return descendantsNAire (arbre->racine, arbre->toString);
}
```

3.2.9.c Parcours indenté n-aire

Le parcours indenté de l'arbre binaire de la Figure 58 conduit au résultat suivant (voir fonction *indentationPrefixee()*, § 3.2.4.d, page 119) qui peut être intéressant pour la mise au point des diverses fonctions basées sur la mémorisation de cet arbre binaire mais n'a pas de signification particulière pour l'application.

```
Parcours préfixé (avec indentation binaire)
Julie
     Jonatan
          Pauline
               Sonia
                    Paul
          Gontran
               Antonine
```

L'indentation n-aire est plus proche de l'application (voir Figure 57), et affiche les fils d'un nœud avec le même décalage comme indiqué ci-dessous. Il faut explorer le SAG du nœud de départ. C'est un parcours d'arbre avec changement de niveau quand on descend dans le SAG (vers le premier fils). Le SAD concerne les frères qui sont au même niveau.

```
Indentation n-aire
Julie
     Jonatan
          Pauline
          Sonia
          Paul
     Gontran
          Antonine
```

```
// parcours n-aire indenté
static void indentationNAire (Noeud* racine, char* (*toString) (Objet*),
                                                            int niveau) {
  if (racine != NULL) {
    for (int i=1; i<niveau; i++) printf ("%5s", " ");
    printf ("%s\n", toString (racine->reference));
    indentationNAire (racine->gauche, toString, niveau+1);
    indentationNAire (racine->droite, toString, niveau);
  }
}

void indentationNAire (Arbre* arbre) {
  if (arbre->racine != NULL) {
    Noeud* nrac   = cNd (arbre->racine->reference, arbre->racine->gauche,
                                                                   NULL);
    Arbre* narbre = creerArbre (nrac, arbre->toString, NULL);
    indentationNAire (narbre->racine, arbre->toString, 1);
    free (nrac);
    free (narbre);
  }
}
```

3.2.9.d Ascendants n-aires

La fonction *ascendantsNAire()* énumère les ascendants sur l'arbre n-aire d'un nœud donné. ascendantsNaire (racine, “Paul”) fournit *Jonatan Julie*. Sur l'arbre n-aire (voir Figure 57), l'ascendant direct de *Paul* est *Jonatan*, l'ascendant direct de *Jonatan* est *Julie*. Il faut faire un parcours d'arbre interrompu quand on a trouvé l'objet cherché et écrire uniquement quand on remonte d'un SAG (voir Figure 58), donc après l'appel récursif examinant le SAG. La fonction fournit vrai si on a trouvé, faux sinon. Cette fonction est très proche de la fonction *trouverNoeud()* qui elle, fournit un pointeur sur le nœud recherché. La fonction utilise deux pointeurs de fonctions : un pour écrire les caractéristiques des nœuds et l'autre pour trouver le nœud dont on veut les ascendants.

```
// fournir les ascendants n-aires de "objet"
static booleen ascendantsNAire (Noeud* racine, Objet* objet,
                                          char* (*toString) (Objet*),
                                  int (*comparer) (Objet*, Objet*)) {
  booleen trouve;

  if (racine == NULL) {
    trouve = faux;
  } else if ( comparer (objet, racine->reference) == 0 ) {
    printf ("%s ", toString (racine->reference));
    trouve = vrai;
  } else {
    trouve = ascendantsNAire (racine->gauche, objet, toString, comparer);
    if (trouve) {
      printf ("%s ", toString (racine->reference));
    } else {
      trouve = ascendantsNAire (racine->droite, objet, toString,
                                                            comparer);
    }
  }
  return trouve;
}
```

```
booleen ascendantsNAire (Arbre* arbre, Objet* objet) {
  return ascendantsNAire (arbre->racine, objet,
                                        arbre->toString, arbre->comparer);
}
```

3.2.9.e Parcours en largeur n-aire

Le parcours en largeur consiste à traiter les nœuds de l'arbre étage par étage en partant de la racine. Ainsi, sur l'arbre n-aire de la Figure 57, l'énumération en largeur fournit : *Julie, Jonatan Gontran, Pauline Sonia Paul Antonine*. On peut également réécrire le nom du sous-arbre et obtenir un fichier décrivant l'arbre n-aire sous la forme suivante, ce que fait le programme *enLargeurNAire()* ci-dessous.

```
Julie: Jonatan Gontran;
Jonatan: Pauline Sonia Paul;
Gontran: Antonine;
```

La méthode consiste à créer une liste, chaque élément de la liste pointant sur un nœud de l'arbre. Au début, la liste li contient un seul élément pointant sur la racine de l'arbre (voir Figire 73). Ensuite, tant que la liste n'est pas vide, on extrait l'élément en tête de la liste li, et on insère en fin de liste les fils n-aires du nœud. Sur l'exemple, on extrait l'élément de la liste pointant sur *Julie* et on insère dans la liste devenue vide, les fils *n-aires* de *Julie*, soit *Jonatan* et *Gontran*. Au tour suivant, *Jonatan* est remplacé par ses fils n-aires *Pauline, Sonia, Paul*, et *Gontran* est remplacé par *Antonine*. Là également, on utilise la mémorisation de l'arbre binaire pour répondre à des questions de l'arbre n-aire.

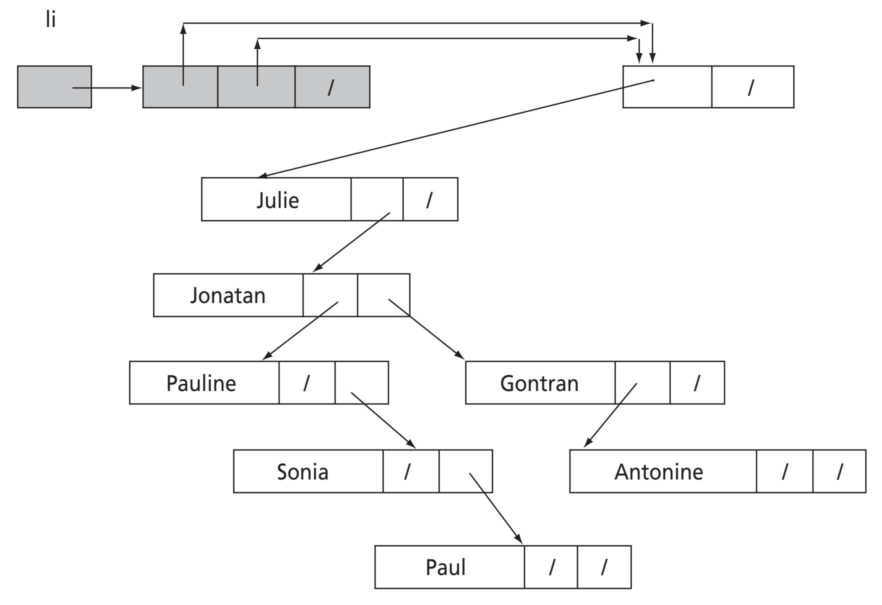


Figure 73 Parcours en largeur n-aire d'un arbre binaire.

```
static void enLargeurNAire (Noeud* racine,  char* (*toString) (Objet*)) {
  Liste* li = creerListe();
  insererEnFinDeListe (li, racine);
  while (!listeVide (li) ) {
    Noeud* extrait   = (Noeud*) extraireEnTeteDeListe (li);
    Noeud* p1F = extrait->gauche;  // premier fils
    if (p1F != NULL) printf ("%s: ", toString (extrait->reference));
    Noeud* pNF = p1F;
    while (pNF != NULL) {
      printf (" %s", toString (pNF->reference));
      insererEnFinDeListe (li, pNF);
      pNF = pNF->droite;
    }
    if (p1F != NULL) printf (";\n");
  }
}

void enLargeurNAire (Arbre* arbre) {
  if (arbre->racine != NULL) {
    Noeud* nrac   = cNd (arbre->racine->reference, arbre->racine->gauche,
                                                                   NULL);
    Arbre* narbre = creerArbre (nrac, arbre->toString, NULL);
    enLargeurNAire (narbre->racine, arbre->toString);
    free (nrac);
    free (narbre);
  }
}
```

3.2.9.f Duplication d'un arbre n-aire sur N niveaux

La fonction *dupArbreNAireSNNiv()* duplique nbniveau d'un arbre n-aire à partir du nœud racine en utilisant la mémorisation de l'arbre binaire. Ainsi, sur l'arbre n-aire de la Figure 57, on peut ne mémoriser que 2 niveaux soit *Julie* et *Jonatan-Gontran*. L'algorithme est un algorithme de duplication d'arbre, y compris des objets référencés (voir § 3.2.6, page 125), le niveau nbniveau étant décrémenté quand on descend dans un SAG vers le premier fils. Il y a arrêt des appels récursifs si l'arbre est vide ou si *nbniveau* vaut 0.

```
static Noeud* dupArbreNAireSNNiv (Noeud* racine, int nbniveau,
                                              Objet* (*cloner) (Objet*)) {
  if ((racine == NULL) || (nbniveau==0) ) {
    return NULL;
  } else {
    Noeud* nouveau = cNd (cloner (racine->reference));
    nouveau->gauche = dupArbreNAireSNNiv (racine->gauche, nbniveau-1,
                                                                 cloner);
    nouveau->droite = dupArbreNAireSNNiv (racine->droite, nbniveau, cloner);
    return nouveau;
  }
}

Arbre* dupArbreNAireSNNiv (Arbre* arbre, int nbniveau,
                                              Objet* (*cloner) (Objet*)) {
  Noeud* racine = dupArbreNAireSNNiv (arbre->racine, nbniveau, cloner);
  return creerArbre (racine);
}
```

3.2.9.g Dessin d'un arbre n-aire

La fonction *dessinerArbreNAire (Noeud* racine, FILE* fs) ;* dessine un arbre en mettant en évidence les successeurs n-aires d'un nœud comme l'indique le schéma de la Figure 74.

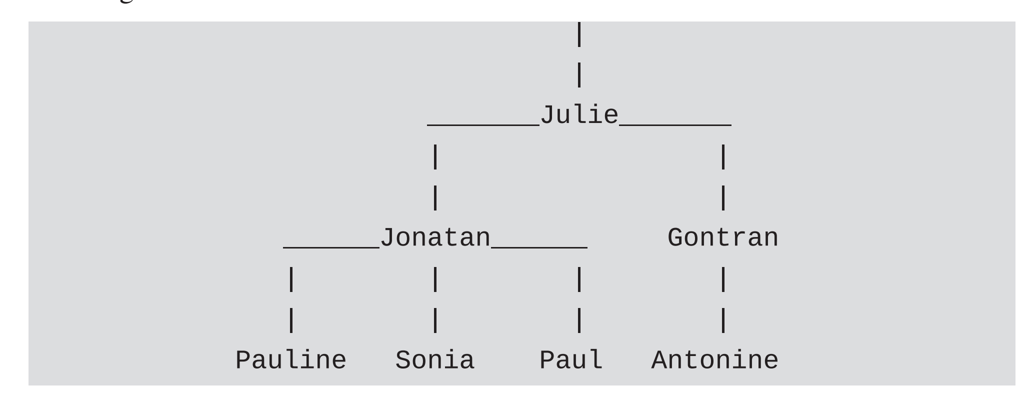


Figure 74 Dessin n-aire d'un arbre.

L'arbre à dessiner est dupliqué en ajoutant pour chaque nœud sa position (son numéro de colonne) sur la feuille de dessin. Les feuilles n-aires de l'arbre sont numérotées dans un parcours infixé : *Pauline* 1, *Sonia* 2, *Paul* 3, *Antonine* 4. La position d'un nœud qui n'est pas une feuille n-aire est la moyenne de la position de son premier et dernier fils n-aires. Ainsi *Jonatan* se trouve en position (1+3)/2 = 2. *Gontran* est en position 4. *Julie* est en position (2+4)/2=3. Cette position est multipliée par la valeur du plus long identificateur déterminé par la fonction maxIdent().

On effectue alors un parcours en largeur. Connaissant les positions (numéros de colonne) de chaque nœud, l'algorithme procède de la même manière que pour le dessin de l'arbre binaire (voir § 3.2.8, page 127). On insère dans une liste lc un pointeur sur la racine de l'arbre à dessiner. On dessine les éléments de la liste courante lc, et on les remplace par leurs fils n-aires dans une liste des suivants ls qui devient la liste courante pour le prochain tour. Les tirets entourant le nom sont écrits entre la position du premier fils et du dernier fils n-aires.

Exercice 16 - Dessin n-aire d'un arbre

En utilisant le module de gestion de listes (voir § 2.3.8, page 48), et en vous inspirant de l'algorithme *dessinerArbre()* (voir § 3.2.8, page 127), écrire :

- la fonction : *static Arbre* dupArbN (Arbre* arbre, int lgM) ;* qui duplique l'arbre et calcule la position de chaque nœud de l'arbre. La fonction *maxIdent()* (voir § 3.2.5.c, page 123) permet de calculer la valeur lgM (largeur maximale d'une colonne).
- la fonction : *void dessinerArbreNAire (Arbre* arbre, FILE* fs) ;* qui dessine l'arbre comme indiqué sur la Figure 74.

3.2.10 Le module des arbres binaires

3.2.10.a Le fichier d'en-tête pour les arbres binaires

Ce fichier d'en-tête *arbre.h* contient les définitions et les prototypes des fonctions concernant les arbres binaires et les questions n-aires sur ces arbres binaires comme vu précédemment.

```
// arbre.h

#ifndef ARBRE_H
#define ARBRE_H

typedef int  booleen;
#define faux 0
#define vrai 1

typedef void Objet;

typedef struct noeud {
  Objet*        reference;
  struct noeud* gauche;
  struct noeud* droite;
  int    factEq;    // facteur d'équilibre : si arbre équilibré
} Noeud;

typedef struct {
  Noeud* racine;
  char* (*toString) (Objet*);
  int   (*comparer) (Objet*, Objet*);
} Arbre;


Noeud* getracine             (Arbre* arbre);
Objet* getobjet              (Noeud* noeud);
Noeud* getsag                (Noeud* noeud);
Noeud* getsad                (Noeud* noeud);
void   setracine             (Arbre* arbre, Noeud* racine);
void   settoString           (Arbre* arbre, char* (*toString) (Objet*));
void   setcomparer           (Arbre* arbre,
                                       int (*comparer) (Objet*, Objet*));

// création de noeuds
Noeud*   cNd                (Objet* objet, Noeud* Gauche, Noeud* Droite);
Noeud*   cNd                 (Objet* objet);
Noeud*   cF                  (Objet* objet);

// création d'arbre
void     initArbre           (Arbre* arbre, Noeud* racine,
           char* (*toString) (Objet*), int (*comparer) (Objet*, Objet*));
Arbre*   creerArbre          (Noeud* racine,
           char* (*toString) (Objet*), int (*comparer) (Objet*, Objet*));
Arbre*   creerArbre          (Noeud* racine);
Arbre*   creerArbre          ();

// parcours
void     prefixe             (Arbre* arbre);
void     infixe              (Arbre* arbre);
void     postfixe            (Arbre* arbre);
void     infixeDG            (Arbre* arbre);
```

```
void     infixe               (Arbre* arbre, void (*f) (Objet*));
void     indentationPrefixee  (Arbre* arbre);
void     indentationPostfixee (Arbre* arbre);
Noeud*   trouverNoeud         (Arbre* arbre, Objet* objet);
void     enLargeur            (Arbre* arbre);
void     enLargeurParEtage    (Arbre* arbre);

// propriétés
int      taille               (Noeud* noeud);
int      taille               (Arbre* arbre);
booleen  estFeuille           (Noeud* arbre);
int      nbFeuilles           (Arbre* arbre);
void     listerFeuilles       (Arbre* arbre);
int      maxIdent             (Arbre* arbre);
int      somLgIdent           (Arbre* arbre);
int      hauteur              (Arbre* arbre);
booleen  degenere             (Arbre* arbre);
booleen  equilibre            (Arbre* arbre);

// duplication, destruction, dessin
Arbre*   dupliquerArbre       (Arbre*  arbre);
Arbre*   dupliquerArbre       (Arbre*  arbre, Objet* (*cloner) (Objet*));
void     detruireArbre        (Arbre*  arbre);
void     detruireArbre        (Arbre*  arbre,
                                  void (*detruireObjet) (Objet*));
void     dessinerArbre        (Arbre*  racine,  FILE* fs);
booleen  egaliteArbre         (Arbre* arbre1, Arbre* arbre2);

// binaire NAire
int      nbFeuillesNAire      (Arbre* arbre);
void     listerFeuillesNAire  (Arbre* arbre);
void     descendantsNAire     (Arbre* arbre);
void     indentationNAire     (Arbre* arbre);
booleen  ascendantsNAire      (Arbre* arbre, Objet* objet);
void     enLargeurNAire       (Arbre* racine);
Arbre*   dupArbreNAireSNNiv   (Arbre* arbre, int niveau,
                                       Objet* (*cloner) (Objet*));
void     dessinerArbreNAire   (Arbre* arbre, FILE* fs);

// arbre ordonné                                            § 3.3, page 156
void     insererArbreOrd      (Arbre* arbre, Objet* objet);
Noeud*   supprimerArbreOrd    (Arbre* arbre, Objet* objet);
Noeud*   rechercherOrd        (Arbre* arbre, Objet* objet);
Objet*   minArbreOrd          (Arbre* arbre);
Objet*   maxArbreOrd          (Arbre* arbre);

// arbre AVL                                                § 3.4, page 167
void     insererArbreEquilibre (Arbre* arbre, Objet* objet);


// arbre de chaînes de caractères (cas particulier)    § 3.2.11, page 140
// création de noeuds
// pour cF() et cNd(), la chaîne de caractères n'est pas dupliquée
// pour cFCh(), la chaîne est dupliquée
Noeud* cF            (char* message);
Noeud* cNd           (char* message, Noeud* gauche, Noeud* droite);
Noeud* cFCh          (char* message);
// créer un arbre de chaînes de caractères à partir d'un fichier
Arbre* creerArbreCar (FILE* fe);

#endif
```

3.2.10.b Le corps du module des arbres binaires

```
// arbre.cpp module de création et manipulation d'arbres

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "arbre.h"

// fournir la valeur de la racine de l'arbre
Noeud* getracine (Arbre* arbre) {
  return arbre->racine;
}

// modifier la valeur de la racine de l'arbre
void setracine (Arbre* arbre, Noeud* racine) {
  arbre->racine = racine;
}

// fournir l'objet d'un noeud
Objet* getobjet (Noeud* racine) {
  return racine->reference;
}

// fournir un pointeur sur le sag
Noeud* getsag (Noeud* noeud) {
  return noeud->gauche;
}

// fournir un pointeur sur le sad
Noeud* getsad (Noeud* noeud) {
  return noeud->droite;
}

// modifier la fonction toString de arbre
void settoString (Arbre* arbre, char* (*toString) (Objet*)) {
  arbre->toString = toString;
}

// modifier la fonction comparer de arbre
void setcomparer (Arbre* arbre, int (*comparer) (Objet*, Objet*)) {
  arbre ->comparer = comparer;
}


booleen arbreVide (Arbre* arbre) {
  return arbre->racine == NULL;
}

Les pages précédentes ont détaillé le corps des procédures suivantes à
insérer ici :
cNd, cF, initArbre, creerArbre, prefixe, infixe, postfixe, indentationPre-
fixee, trouverNoeud, enLargeur, enLargeurParEtage,
taille, estFeuille, nbFeuilles, listerFeuilles, maxIdent, somLgIdent,
hauteur, degenere, dupliquerArbre, detruireArbre, egaliteArbre, dessiner-
Arbre,
listerFeuillesNAire, nbFeuillesNAire, descendantsNAire, indentation-
NAire, ascendantsNAire, enLargeurNAire, dupArbreNAireSNNiv,
dessinerArbreNAire (à faire voir exercice 16, page 136)
```

Les fonctions sur les arbres ordonnés et sur les arbres AVL sont données dans les paragraphes suivants.

3.2.11 Les arbres de chaînes de caractères

Les arbres de chaînes de caractères sont un cas particulier d'arbres où chaque nœud référence un objet chaîne de caractères.

Les fonctions *creerArbreGene()* (voir § 3.2.3.a, page 111) et *creerArbreExp()* (voir § 3.2.3.b, page 112) effectuent la construction respectivement des exemples de l'arbre généalogique et de l'arbre de l'expression arithmétique ce qui permet de tester les diverses fonctions sur les arbres binaires vues précédemment. Pour changer d'arbre, il faut réécrire ces fonctions. Si l'arbre est volumineux, l'écriture devient vite complexe. La fonction *creerArbreCar()* construit un arbre binaire à partir d'une description n-aire de l'arbre donnée dans un fichier. L'arbre n-aire de la Figure 57, page 109, se note comme suit dans un fichier nommé *Julie.nai* :

```
Julie:      Jonatan  Gontran;
Jonatan:    Pauline  Sonia     Paul;
Gontran:    Antonine;
```

Julie a deux enfants *Jonatan* et *Gontran* ; *Jonatan* a trois enfants *Pauline*, *Sonia*, *Paul*. Cette description correspond à l'arbre n-aire de la Figure 57. À partir de cette description la fonction *creerArbreCar()* crée l'arbre binaire équivalent schématisé sur la Figure 58.

Les fonctions suivantes *lireUnMot()*, *ajouterFils()* et *creerArbre()* créent l'arbre binaire à partir d'une description n-aire contenue dans un fichier pour des nœuds contenant un objet de type chaîne de caractères. (voir la syntaxe de *homme.nai* dans l'exercice 17 ou de *terre.nai* § 3.2.11.b, page 147). Il faut donc faire un petit analyseur reconnaissant les données conformes à la grammaire de description d'un arbre n-aire. *lireUnMot()* lit une chaîne de caractères du fichier fe, en ignorant les blancs en tête du mot, jusqu'à trouver un délimiteur de mots soit un espace, `':'` ou `';'`. *ajouterFils()* ajoute les fils n-aires au nœud pere. *trouverNoeud()* (voir § 3.2.4.e, page 119) fournit un pointeur sur le nœud pere s'il existe dans l'arbre, sinon fournit NULL. On ajoute le premier fils dans le SAG de pere et on chaîne entre eux les fils suivants dans le champ droite du premier fils. Le dernier fils est suivi de `';'`. *creerArbreCar()* lit le nom du pere (suivi de `':'`) et appelle *ajouterFils()* pour traiter les fils de pere.

```
// arbre de chaînes de caractères

Noeud* cF (char* message) {
  return cF ( (Objet*) message);
}

Noeud* cNd (char* message, Noeud* gauche, Noeud* droite) {
  return cNd ( (Objet*) message, gauche, droite);
}
```

```
// créer une feuille contenant un objet "chaîne de caractères";
// la chaîne de caractères est dupliquée
Noeud* cFCh (char* objet) {
  char* nobjet = (char*) malloc (sizeof (strlen(objet)+1));
  strcpy (nobjet, objet);
  return cF (nobjet);
}

// fournir la chaîne de caractères de objet
static char* toChar (Objet* objet) {
  return (char*) objet;
}

// comparer deux chaînes de caractères
// fournit <0 si ch1 < ch2; 0 si ch1=ch2; >0 sinon
static int comparerCar (Objet* objet1, Objet* objet2) {
  return strcmp ((char*)objet1, (char*)objet2);
}

static Objet* clonerCar (Objet* objet) {
  char* message = (char*) objet;
  int lg = strlen (message);
  char* nouveau = (char*) malloc (lg+1);
  strcpy (nouveau, message);
  return (Objet*) nouveau;
}

char c; // prochain caractère à analyser dans le fichier
        // fe de description de l'arbre

// lire dans chaine un mot du fichier fe
static void lireUnMot (FILE* fe, char* chaine) {
  // ignorer les espaces en tête du mot
  while ( ((c==' ') || (c=='\n') || (c=='\r')) && !feof (fe) ) {
    fscanf (fe, "%c", &c);
  }

  // enregistrer les caractères jusqu'à trouver ' ', ':' ou ';'
  char* pCh = chaine;
  while ( (c != ' ') && (c!=':') && (c!=';') ) {
    *pCh++ = c;
    fscanf (fe, "%c", &c);
    if (c=='\n') c= ' ';
  }
  *pCh = 0;
}

// ajouter un ou plusieurs fils n-aire au noeud pere
static void ajouterFils (FILE* fe, Arbre* arbre, char* pere) {
  char nom [255];

  Noeud* pNom = trouverNoeud (arbre, pere);
  if (pNom != NULL) {
    // lire le premier fils de pere
    fscanf (fe, "%c", &c);  // passer le délimiteur :
    lireUnMot (fe, nom);
    Noeud* fils = cFCh (nom);
    pNom->gauche = fils;
```

```
      // lire les fils suivants de pere jusqu'à ';'
      Noeud* filsPrec = fils;
      while ( (c!=';') && !feof (fe) ) {     // après ;
        fscanf (fe, "%c", &c);  // lit le délimiteur espace
        lireUnMot (fe, nom);
        fils            = cFCh (nom);
        filsPrec->droite = fils;
        filsPrec        = fils;
      }
    } else {
       printf ("Noeud %s non trouvé\n", pere);
    }
  }

  // créer un arbre de chaînes de caractères
  // à partir d'un fichier n-aire
  Arbre* creerArbreCar (FILE* fe) {
    Arbre*  arbre  = creerArbre();
    booleen debut  = vrai;
    fscanf (fe, "%c", &c);
    char pere [255];
    lireUnMot (fe, pere);

    while (!feof(fe)) {
      if (debut) {
        setracine (arbre, cFCh (pere));
        debut = faux;
      }
      ajouterFils (fe, arbre, pere);
      fscanf (fe, "%c", &c);    // passer le délimiteur ;
      lireUnMot (fe, pere);
    }
    return arbre;
  }
```

Exercice 17 - Le corps humain

Soit le fichier *homme.nai* suivant :

```
homme:   tete cou tronc bras jambe;
tete:    crane yeux oreille cheveux bouche;
tronc:   abdomen thorax;
thorax:  coeur foie poumon;
jambe:   cuisse mollet pied;
pied:    cou-de-pied orteil;
bras:    epaule avant-bras main;
main:    doigt;
```

Dessiner l'arbre n-aire et l'arbre binaire correspondant.

3.2.11.a Menu de test du module des arbres binaires

Dans le programme de test des arbres binaires suivant, il y a un arbre par défaut (l'arbre généalogique) qui peut être changé (choix 2 pour avoir l'arbre de l'expression arithmétique, ou choix 17 pour construire l'arbre à partir d'un fichier). La racine de l'arbre est alors le nœud courant. L'option 3 permet de changer ce nœud courant. Les différentes fonctions s'exécutent en partant de ce nœud courant

(parcours, dessins, etc.). Le menu proposé permet de tester les différentes fonctions vues précédemment concernant l'arbre binaire ou les interrogations n-aires.

```
// pparbre.cpp  programme principal arbre

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>  //isalpha
#include "arbre.h"
#include "mdtypes.h"
#include "arbrstat.h"

// créer un arbre binaire généalogique
Arbre* creerArbreGene () {
  Noeud* racine =
    cNd ( "Julie",
          cNd ( "Jonatan",
                cNd ( "Pauline",
                      NULL,
                      cNd ( "Sonia", NULL, cF ("Paul") )
                    ),
                cNd ( "Gontran", cF ("Antonine"), NULL)
               ),
          NULL
        );
  return creerArbre (racine);
}


// créer un arbre binaire (expression arithmétique)
Arbre* creerArbreExp () {
  Noeud* racine =
    cNd ( "-",
          cNd ( "*",
                cNd ("+", cF ("a"), cF ("b") ),
                cNd ("-", cF ("c"), cF ("d") )
              ),
          cF  ("e")
        );
  return creerArbre (racine);
}

int menu (Arbre* arbre) {
  printf ("\n\nARBRES BINAIRES\n\n");
  printf (" 0 - Fin du programme\n");
  printf ("\n");
  printf (" 1 - Création de l'arbre généalogique\n");
  printf (" 2 - Création de l'arbre de l'expression arithmétique\n");
  printf (" 3 - Nom du noeud courant (défaut:racine)\n");
  printf ("\n");
  printf (" 4 - Parcours préfixé\n");
  printf (" 5 - Parcours infixé\n");
  printf (" 6 - Parcours postfixé\n");
  printf (" 7 - Parcours préfixé (avec indentation)\n");
  printf (" 8 - Parcours en largeur\n");
  printf ("\n");
  printf (" 9 - Taille  et longueur du plus long identificateur\n");
  printf ("10 - Nombre  et liste des feuilles\n");
  printf ("11 - Hauteur de l'arbre binaire\n");
  printf ("12 - Tests   arbre dégénéré ou équilibré\n");
```

```
  printf ("\n");
  printf ("13 - Duplication de l'arbre\n");
  printf ("14 - Destruction de l'arbre dupliqué\n");
  printf ("\n");
  printf ("15 - Dessin de l'arbre binaire (écran)\n");
  printf ("16 - Dessin de l'arbre binaire (fichier)\n");
  printf ("\n");
  printf ("Arbres n-aires\n");
  printf ("17 - Création à partir d'un fichier n-aire\n");
  printf ("18 - Indentation            n-aire\n");
  printf ("19 - Descendants            n-aires\n");
  printf ("20 - Ascendants             n-aires\n");
  printf ("21 - Feuilles               n-aires\n");
  printf ("22 - Parcours en largeur    n-aire\n");
  printf ("23 - Dessin des descendants (écran)\n");
  printf ("24 - Dessin des descendants (fichier)\n");
  printf ("25 - Nombre de niveaux      utilisés\n");
  printf ("26 - Arbre statique         \n");

  printf ("\n");
  Noeud* racine = getracine (arbre);
  if (racine!=NULL) printf ("Noeud courant : %s\n",
    arbre->toString (racine->reference));
  fprintf (stderr, "Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  printf ("\n");

  return cod;
}

void main () {
  Arbre* arbre    = creerArbreGene();
  Arbre* arbreBis = creerArbre();

  booleen fini = faux;
  while (!fini) {
    switch (menu (arbre) ) {

    case 0:
      fini = vrai;
      break;

    case 1:
      printf ("Création de l'arbre généalogique\n");
      detruireArbre (arbre);
      arbre = creerArbreGene();
      break;

    case 2:
      printf ("Création de l'arbre expr. arithmét.\n");
      detruireArbre (arbre);
      arbre = creerArbreExp();
      break;

    case 3: {
      printf ("Nom du noeud courant ? ");
      ch15 nom; scanf ("%s", nom);
      Noeud* trouve = trouverNoeud (arbre, nom);
      if (trouve == NULL) {
        printf ("%s inconnu dans l'arbre\n", nom);
      } else {
        arbre = creerArbre (trouve);
```

```
        }
        } break;

    case 4: {
        printf ("Parcours préfixé\n");
        prefixe (arbre);
        } break;

    case 5:
        printf ("Parcours infixé\n");
        infixe (arbre);
        break;

    case 6:
        printf ("Parcours postfixé\n");
        postfixe (arbre);
        break;

    case 7:
        printf ("Parcours préfixé (avec indentation)\n");
        indentationPrefixee (arbre);
        break;

    case 8:
        printf ("Parcours en largeur\n");
        enLargeur (arbre);
        printf ("\n\nParcours en largeur par étage\n");
        enLargeurParEtage (arbre);
        break;

    case 9:
        printf ("Taille de l'arbre : %d\n", taille (arbre) );
        printf ("Longueur de l'ident. le plus long %d\n", maxIdent (arbre));
        break;

    case 10:
        printf ("Nombre de feuilles : %d\n", nbFeuilles (arbre) );
        printf ("Liste des feuilles : ");
        listerFeuilles (arbre);
        break;

    case 11:
        printf ("Hauteur de l'arbre : %d\n", hauteur (arbre) );
        break;

    case 12:
        if (degenere (arbre)) {
          printf ("Arbre dégénéré\n");
        } else {
          printf ("Arbre non dégénéré\n");
        }
        if (equilibre (arbre) ) {
          printf ("Arbre équilibré\n");
        } else {
          printf ("Arbre non équilibré\n");
        }
        break;

    case 13:
        arbreBis = dupliquerArbre (arbre);
        //arbreBis = dupliquerArbre (arbre, clonerCar);
        printf ("Parcours préfixé de l'arbre dupliqué\n");
        prefixe (arbreBis);
```

```
      break;

    case 14:
      detruireArbre (arbreBis);
      if (getracine(arbreBis) == NULL) printf ("Arbre détruit\n");
      break;

    case 15:
      dessinerArbre (arbre, stdout);
      break;

    case 16: {
      printf ("Dessin d'un arbre binaire (fichier)\n");
      printf ("Donner le nom du fichier à créer ? ");
      char nomFS [50]; scanf ("%s", nomFS);
      FILE* fs = fopen (nomFS, "w");
      if (fs==NULL) {
        printf ("%s erreur ouverture\n", nomFS);
      } else {
        dessinerArbre (arbre, fs);
        fclose (fs);
      }
      } break;

    case 17: {
      printf ("Création d'un arbre à partir d'un fichier\n");
      printf ("Nom du fichier décrivant l'arbre n-aire ? ");
      char nomFE [50]; scanf  ("%s", nomFE);
      FILE* fe = fopen (nomFE, "r");
      if (fe==NULL) {
        printf ("%s erreur ouverture\n", nomFE);
      } else {
        detruireArbre (arbre);
        arbre = creerArbreCar (fe);
      }
      } break;

    case 18:
      printf ("Indentation n-aire\n");
      indentationNAire (arbre);
      break;

    case 19:
      printf ("Descendants n-aires\n");
      descendantsNAire (arbre);
      break;

    case 20: {
      printf ("Nom de l'élément dont on veut les ascendants ? ");
      ch15 nom; scanf ("%s", nom); getchar();
      printf ("Ascendants  n-aires\n");
      ascendantsNAire (arbre, nom);
      } break;

    case 21:
      printf ("Feuilles n-aires\n");
      listerFeuillesNAire (arbre);
      break;

    case 22:
      printf ("Parcours en largeur n-aire\n");
      enLargeurNAire (arbre);
      break;
```

```
    case 23: {
      dessinerArbreNAire (arbre, stdout);
      } break;

    case 24: {
      printf ("Dessin d'un arbre n-aire (fichier)\n");
      printf ("Donner le nom du fichier à créer ? ");
      char nomFS [50]; scanf ("%s", nomFS);
      FILE* fs = fopen (nomFS, "w");
      if (fs==NULL) {
        printf ("%s erreur ouverture\n", nomFS);
      } else {
        dessinerArbreNAire (arbre, fs);
        fclose (fs);
      }
      } break;

    case 25: {
      printf ("Nombre de niveaux à considérer\n");
      printf ("dans l'arbre n-aire ? ");
      int nbNiv; scanf  ("%d", &nbNiv); getchar();
      Arbre* sa  = dupArbreNAireSNNiv (arbre, nbNiv, clonerCar);
      dessinerArbreNAire (sa, stdout);
      } break;

    case 26: {                                              voir 3.2.1.2
      ArbreS* arbres = creerArbreStat (arbre);
      printf ("\nNombre de noeuds (arbre en tableau) : %d\n",
                                                 tailleStat (arbres));
      printf ("\nEcriture du tableau\n");
      ecrireStat (arbres);
      printf ("\nIndentation n-aire en utilisant le tableau\n");
      indentationNAireStat (arbres);
      Arbre* arbre2 = creerArbreDyn (arbres);
      dessinerArbreNAire (arbre2, stdout);
      } break;

    }  // switch
    if (!fini) {
      printf ("\n\nTaper Return pour continuer\n"); getchar();
    }
  } // while

}
```

3.2.11.b Exemples de créations et d'interrogations d'arbres binaires

Les encadrés suivants présentent des résultats de tests du programme de gestion des arbres binaires. Le choix 17 a permis la construction de l'arbre à partir du fichier *terre.nai*. Le choix 3 définit le nœud *Europe* comme racine du sous-arbre par défaut. Le choix 23 dessine l'arbre à partir du nœud courant *Europe*.

Le fichier *terre.nai* (vous pouvez compléter ... jusqu'à votre bar favori !) :

```
Terre:      Europe    Asie      Afrique    Amerique  Oceanie;
Europe:     France    Espagne   Belgique   Danemark;
France:     Bretagne  Corse     Bourgogne;
Asie:       Chine     Inde      Irak       Japon;
Afrique:    Niger     Congo;
```

```
ARBRES BINAIRES

 0 - Fin du programme

 1 - Création de l'arbre généalogique
 2 - Création de l'arbre de l'expression arithmétique
 3 - Nom du noeud courant (défaut:racine)

 4 - Parcours préfixé
 5 - Parcours infixé
 6 - Parcours postfixé
 7 - Parcours préfixé (avec indentation)
 8 - Parcours en largeur

 9 - Taille  et longueur du plus long identificateur
10 - Nombre  et liste des feuilles
11 - Hauteur de l'arbre binaire
12 - Tests   arbre dégénéré ou équilibré

13 - Duplication de l'arbre
14 - Destruction de l'arbre dupliqué

15 - Dessin de l'arbre binaire (écran)
16 - Dessin de l'arbre binaire (fichier)

Arbres n-aires
17 - Création à partir d'un fichier n-aire
18 - Indentation            n-aire
19 - Descendants            n-aires
20 - Ascendants             n-aires
21 - Feuilles               n-aires
22 - Parcours en largeur    n-aire
23 - Dessin des descendants (écran)
24 - Dessin des descendants (fichier)
25 - Nombre de niveaux      utilisés

Noeud courant : Europe
Votre choix ? 23
```

```
                                          |
                                          |
                          _________________Europe__________________
                          |                    |         |         |
                          |                    |         |         |
               _______France________        Espagne  Belgique  Danemark
               |          |         |
               |          |         |
           Bretagne     Corse    Bourgogne

       Taper Return pour continuer
```

Autre exemple d'interrogation : l'indentation n-aire en partant de Europe.

```
       Noeud courant : Europe
       Votre choix ? 18

       Indentation n-aire
       Europe
            France
                 Bretagne
                 Corse
                 Bourgogne
            Espagne
            Belgique
            Danemark

       Taper Return pour continuer
```

3.2.12 Arbre binaire et tableau

Un tableau est habituellement alloué statiquement en fonction de la longueur déclarée à la compilation. On peut cependant aussi allouer un tableau dynamiquement à l'exécution avec *malloc()*. Le terme d'allocation statique pour un tableau devient alors ambigu ; il vaut mieux parler d'allocation contiguë, l'espace mémoire alloué pour le tableau se trouvant sur des cases mémoires contiguës.

3.2.12.a Définitions, déclarations et prototypes de fonctions

L'arbre binaire peut aussi être mémorisé en allocation contiguë (voir Figure 75) soit en mémoire centrale soit sur mémoire secondaire. Cette représentation permet le stockage et la transmission de l'arbre, éventuellement pour recréer la forme dynamique si l'arbre doit évoluer.

Les déclarations d'interface suivantes sont faites dans le fichier d'en-tête *arbrstat.h*. *NœudS* est la structure correspondant à une ligne du tableau. *creerAr-*

brStat() alloue de l'espace pour un tableau à créer à partir de sa représentation dynamique ; la taille du tableau est donnée par la fonction *taille()* qui opère sur la représentation dynamique de l'arbre. *creerArbreStat()* remplit le tableau en faisant un parcours d'arbre : on crée la structure de la Figure 75 à partir de celle de la Figure 58, page 109 ; la racine de l'arbre se trouve dans l'entrée 0. *creerArbreDyn()* effectue la conversion inverse de tableau en allocation dynamique. Si l'arbre doit être modifié (ajout et retrait), il est plus facile d'utiliser la version avec allocation dynamique et de regénérer le tableau si besoin est. *tailleStat()* donne la taille de l'arbre du tableau (voir fonction *taille()* § 3.2.5.a, page 122). *ecrireStat()* écrit séquentiellement le contenu du tableau. *indentationNaireStat()* effectue un parcours préfixé indenté de l'arbre du tableau (voir *indentationNaire()*, § 3.2.9.c, page 132).

	nom	**gauche**	**droite**
0	Julie	1	/
1	Jonatan	2	5
2	Pauline	/	3
3	Sonia	/	4
4	Paul	/	/
5	Gontran	6	/
6	Antonine	/	/

Figure 75 Allocation contiguë de l'arbre généalogique.

```
/* arbrstat.h  conversion arbre statique
   en arbre dynamique et vice versa */

#ifndef ARBRSTAT_H
#define ARBRSTAT_H

#include "arbre.h"

#define  NILE -1
typedef  char Chaine [16];

typedef struct {
  Chaine nom;
  int    gauche;
  int    droite;
} NoeudS;

typedef struct {
  NoeudS* as;
} ArbreS;

ArbreS* creerArbreStat       (Arbre*  arbre);
Arbre*  creerArbreDyn        (ArbreS* arbres);
int     tailleStat           (ArbreS* arbres);
void    ecrireStat           (ArbreS* arbres);
```

```
void    indentationNAireStat (ArbreS* arbres);

#endif
```

3.2.12.b Le module des arbres en tableaux

Les fonctions de conversion d'arbres (allocation dynamique ↔ contiguë)

Le fichier arbrstat.cpp contient les corps des fonctions décrites dans *arbrstat.h* et faisant la conversion arbre dynamique arbre statique. Les procédures *creerArbreStat()* et *creerArbreDyn()* sont des fonctions de duplication d'arbres avec changement de la structure de base. Elles sont donc très proches de la fonction *dupliquerArbre()* vue § 3.2.6, page 125.

```
/* arbrstat.cpp  conversion d'arbre dynamique
                 en arbre statique et réciproquement */

#include <stdio.h>
#include <stdlib.h>      // malloc
#include <string.h>      // strcpy
#include "arbrstat.h"    // NoeudS
```

```
// parcours de l'arbre racine en allocation dynamique
// et création du tableau as équivalent
static int creerArbreStat (Noeud* racine, NoeudS* as, int* nf,
                                          char* (toString) (Objet*) ) {
  if (racine == NULL) {
    return -1;
  } else {
    int numNd = *nf;  // numéro du noeud en cours
    *nf = *nf +1;     // nombre total de noeuds
    strcpy (as[numNd].nom, toString (racine->reference));
    as [numNd].gauche = creerArbreStat (racine->gauche, as, nf, toString);
    as [numNd].droite = creerArbreStat (racine->droite, as, nf, toString);
    return numNd;
  }
}

// initialisation et création du tableau as (arbre statique)
ArbreS* creerArbreStat (Arbre* arbre) {
  int     nf = 0;
  ArbreS* arbres = new ArbreS();
  arbres->as     = (NoeudS *) malloc (sizeof(NoeudS) * taille(arbre));
  creerArbreStat (arbre->racine, arbres->as, &nf, arbre->toString);
  return arbres;
}
```

```
// création d'un arbre de chaînes de caractères
// en allocation dynamique en partant du tableau as
static Noeud* creerArbreDyn (NoeudS* as, int racine) {
  if (racine == NILE) {
    return NULL;
  } else {
    Noeud* nouveau  = cF (as[racine].nom);
    nouveau->gauche = creerArbreDyn (as, as[racine].gauche);
    nouveau->droite = creerArbreDyn (as, as[racine].droite);
```

```
    return nouveau;
  }
}

Arbre* creerArbreDyn (ArbreS* arbres) {
  return creerArbre (creerArbreDyn (arbres->as, 0));
}
```

Quelques fonctions sur la structure d'arbre en tableau

On peut facilement réécrire la fonction *taille()* de l'arbre avec la structure sous forme de tableau.

```
// taille de l'arbre statique
static int tailleStat (NoeudS* as, int racine) {
  if (racine == NILE) {
    return 0;
  } else {
    return  1 + tailleStat (as, as[racine].gauche)
              + tailleStat (as, as[racine].droite);
  }
}

int tailleStat (ArbreS* arbres) {
  return tailleStat (arbres->as, 0);
}
```

```
// écriture du tableau
void ecrireStat (ArbreS* arbres) {
  int n = tailleStat (arbres);
  for (int i=0; i<n; i++) {
    NoeudS*  nd = &arbres->as[i];
    printf ("%2d %15s %3d %3d\n", i, nd->nom, nd->gauche, nd->droite);
  }
}
```

```
// indentation n-aire en utilisant le tableau.
// Parcours préfixé du tableau as; racine repère le noeud
// racine du sous-arbre; niveau indique l'indentation
static void indentationNAireStat (NoeudS* as, int racine, int niveau) {
  if (racine != NILE) {
    for (int i=1; i<niveau; i++) printf ("%5s"," ");
    printf ("%s\n", as[racine].nom);

    indentationNAireStat (as, as[racine].gauche, niveau+1);
    indentationNAireStat (as, as[racine].droite, niveau);
  }
}

void indentationNAireStat (ArbreS* arbres) {
  indentationNAireStat (arbres->as, 0, 1);
}
```

3.2.12.c Exemple de conversion allocation dynamique / allocation en tableau

Exemple d'interrogation et de résultats obtenus pour l'arbre généalogique de la Figure 58, page 109. On peut rajouter un 26[e] cas au menu de l'arbre binaire (voir § 3.2.11.a, page 142).

```
Noeud courant : Julie
Votre choix ? 26

Nombre de noeuds (arbre en tableau) : 7

Ecriture du tableau
 0           Julie   1  -1
 1         Jonatan   2   5
 2         Pauline  -1   3
 3           Sonia  -1   4
 4            Paul  -1  -1
 5         Gontran   6  -1
 6        Antonine  -1  -1

Indentation n-aire en utilisant le tableau
Julie
     Jonatan
          Pauline
          Sonia
          Paul
     Gontran
          Antonine
```

3.2.13 Arbre binaire et fichier

Lorsque l'arbre binaire est trop volumineux, il faut le mémoriser sur disque. Ce serait le cas de la nomenclature d'un avion. Un nœud est repéré par son numéro d'enregistrement dans le fichier déclaré de type PNoeud ci-dessous. La fonction *void lireD (int n, Noeud* enr) ;* effectue la lecture dans le fichier fr (variable globale) du nième enregistrement du fichier fr et met cet enregistrement à l'adresse contenue dans enr. La procédure *prefixeF()* effectue un parcours d'arbre en lisant un enregistrement à chaque appel récursif. Il y a autant d'allocations du nœud enr qu'il y a de niveaux d'appels récursifs (voir § 1.2, page 2). La fonction *tailleF()* est également redéfinie ci-dessous pour une allocation dans un fichier en accès direct. L'adaptation des fonctions définies sur les arbres binaires en allocation dynamique à une structure statique dans un fichier ne pose pas de problème particulier (si ce n'est qu'il faut faire des lectures d'enregistrements).

```
#define  NILE   -1

typedef  char   ch15 [16];
typedef  int    PNoeud;

typedef struct noeud {
  ch15   nom;
  PNoeud gauche;
  PNoeud droite;
} Noeud;

FILE* fr;   // fichier binaire contenant l'arbre
```

```
void lireD (int n, Noeud* enr) {
  fseek (fr, (long) n*sizeof (Noeud), 0);
  fread (enr, sizeof (Noeud), 1, fr);
}

// voir § 3.2.4.d, page 117
void prefixeF (PNoeud racine, int niveau) {

  if (racine != NILE) {
    Noeud enr;
    lireD (racine, &enr);

    for (int i=0; i<niveau; i++) printf ("%5s"," ");
    printf ("%s\n", enr.nom);

    prefixeF (enr.gauche, niveau+1);
    prefixeF (enr.droite, niveau+1);
  }
}

int tailleF (PNoeud racine) {
  int t;

  if (racine == NILE) {
    t = 0;
  } else {
    Noeud enr;
    lireD (racine, &enr);
    t = 1 + tailleF (enr.gauche) + tailleF (enr.droite);
  }
  return t;
}
```

3.2.14 Arbre binaire complet

Un arbre binaire est complet si chaque nœud interne a deux successeurs et si toutes les feuilles sont au même niveau.

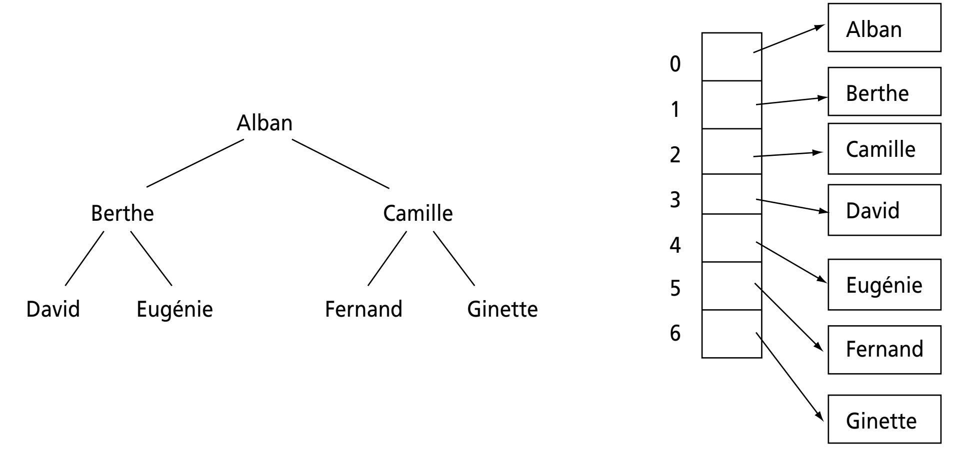


Figure 76 Mémorisation d'un arbre binaire complet.

Aucune place n'est perdue pour mémoriser les pointeurs gauche et droite qui sont implicites. Soit n le nombre de nœuds de l'arbre (n vaut 7 sur la Figure 76). On a alors les relations suivantes :

père (i)	= (i-1) / 2	si 0<i< n, sinon nil
gauche (i)	= 2*i + 1	si 2*i + 1 < n, sinon nil
droite (i)	= 2*i + 2	si 2*i + 2 < n, sinon nil

père (5) = (5-1) / 2 = 2	père ("Fernand") = "Camille"
gauche (2) = 2 * 2 + 1	gauche ("Camille") = "Fernand"
droite (2) = 2 * 2 + 2 = 6	droite ("Camille") = "Ginette"

On peut facilement réécrire les fonctions de parcours avec cette mémorisation compacte. Le tableau tab contient les pointeurs vers les nœuds de l'arbre. `prefixeC()` effectue un parcours préfixé indenté pour un arbre complet. Le parcours séquentiel des éléments du tableau correspond au parcours en largeur d'un arbre.

```
/* acomplet.cpp  arbre complet */

#include <stdio.h>
#define NILE -1

char* tab [] = {"Alban",   "Berthe",  "Camille", "David",
                "Eugénie", "Fernand", "Ginette" };
int   n = 7;   // taille de l'arbre

int gauche (int i) {
  return 2*i+1<n ? 2*i+1 : NILE;
}

int droite (int i) {
  return 2*i+2<n ? 2*i+2 : NILE;
}

// parcours préfixé indenté pour un arbre complet
void prefixeC (int racine, int niveau) {
  if (racine != NILE) {
    for (int i=1; i<niveau; i++) printf ("%5s", " ");
    printf ("%s\n",  tab [racine]);
    prefixeC (gauche(racine), niveau+1);
    prefixeC (droite(racine), niveau+1);
  }
}

void main () {
  // 0 : racine de l'arbre en tab [0];
  // 1 : niveau d'indentation
  prefixeC (0, 1);
}
```

La méthode peut s'appliquer si l'arbre n'est pas parfait, mais il y a perte de place (voir l'exemple de la Figure 77 qui est un cas extrême d'arbre dégénéré). L'arbre nécessite 7 places pour ranger les 3 nœuds. Si on doit faire des insertions ou des suppressions de nœuds, cette représentation d'arbre complet n'est pas pratique.

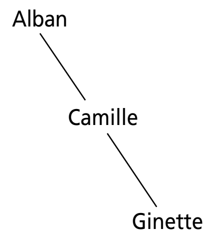


Figure 77 Exemple d'arbre binaire dégénéré.

3.3 LES ARBRES BINAIRES ORDONNÉS

3.3.1 Définitions

Les arbres binaires ordonnés sont des arbres binaires (ayant un SAG et un SAD) tels que pour chaque nœud, les clés (identificateurs) des éléments du sous-arbre gauche sont inférieures à celle de la racine, et celles du sous-arbre droit sont supérieures à celle de la racine. Soit à insérer les clés alphabétiques suivantes dans un arbre ordonné vide et dans l'ordre donné : 17, 11, 15, 14, 31, 19, 18, 28, 26, 35. On aboutit à l'arbre de la Figure 78. La première clé 17 est racine de l'arbre ordonné. Les autres clés sont insérées dans le SAG ou dans le SAD suivant qu'elles sont plus petites ou plus grandes que 17. Chaque nœud d'un arbre ordonné référence un objet contenant une clé et des informations associées à cette clé.

L'arbre peut même constituer un arbre d'index pour un autre fichier. Par exemple, l'information de clé 17 se trouve à l'entrée 125 d'un tableau ou d'un fichier en accès direct. Par la suite, seules les clés sont représentées.

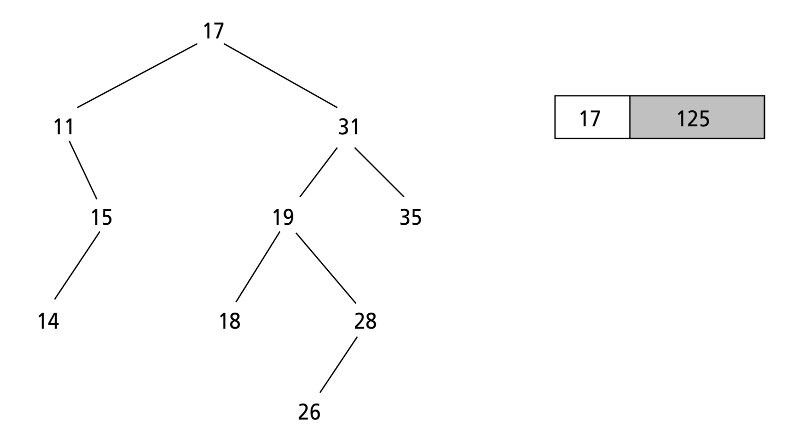


Figure 78 Exemple d'arbre binaire ordonné.

Le parcours infixé gauche-droite de l'arbre donne la liste en ordre croissant : 11, 14, 15, 17, 18, 19, 26, 28, 31, 35. Le parcours infixé droite-gauche donne la liste en ordre décroissant. Ainsi, s'il s'agit d'un fichier de clients, on peut obtenir facilement les clients par ordre croissant ou décroissant de la clé. La recherche d'un élément de l'arbre à partir de sa clé est également facile et rapide. L'insertion se fait toujours au niveau d'une feuille. La suppression d'un nœud feuille ne pose pas de problème mais celle d'un nœud interne demande une réorganisation de l'arbre. L'arbre peut être sur disque. On a alors un index arborescent ordonné sur la clé (numéro ou nom de client par exemple). Les informations sont alors mémorisées dans un fichier de données en accès direct.

Le type arbre ordonné est décrit de la même manière que le type arbre binaire. Il faut cependant définir des fonctions spécifiques de ce type d'arbre concernant l'insertion ou la suppression d'un objet de l'arbre, ou la recherche dans l'arbre en tenant compte du critère d'ordre.

Déclarations faites en fin de `arbre.h` (voir § 3.2.10.a, page 137) et corps des fonctions insérées dans `arbre.cpp` (voir § 3.2.10.b, page 139) :

```
// arbre ordonné
void     insererArbreOrd      (Arbre* arbre, Objet* objet);
Noeud*   supprimerArbreOrd    (Arbre* arbre, Objet* objet);
Noeud*   rechercherOrd        (Arbre* arbre, Objet* objet);
Objet*   minArbreOrd          (Arbre* arbre);
Objet*   maxArbreOrd          (Arbre* arbre);
```

3.3.2 Arbres ordonnés : recherche, ajout, retrait

3.3.2.a Recherche d'un élément dans un arbre binaire ordonné

Il n'est plus nécessaire de parcourir tout l'arbre pour retrouver un élément. À chaque nœud, on peut décider du chemin à suivre, soit dans le sous-arbre gauche, soit dans le sous-arbre droit. Il n'y a pas de retour en arrière pour explorer un autre chemin. Si racine est NULL, on a atteint une feuille sans trouver l'objet cherché, l'élément n'est pas dans l'arbre. Si le nœud racine contient l'objet que l'on cherche, on fournit le pointeur sur le nœud racine. Sinon, si l'objet cherché est inférieur à l'objet de la racine, on explore le SAG, sinon, on explore le SAD. La fonction en paramètre *comparer* est spécifique des objets de l'application, et fournit 0 en cas d'égalité.

```
// fournir un pointeur sur le noeud contenant objet
static Noeud* rechercherOrd (Noeud* racine, Objet* objet,
                                   int (*comparer) (Objet*, Objet*)) {
  int resu;
  if (racine == NULL) {
    return NULL;
  } else if ( (resu=comparer (objet, racine->reference)) == 0 ) {
    return racine;
  } else if (resu < 0) {
    return rechercherOrd (racine->gauche, objet, comparer);
  } else {
```

```
    return rechercherOrd (racine->droite, objet, comparer);
  }
}

Noeud* rechercherOrd (Arbre* arbre, Objet* objet) {
  return rechercherOrd (arbre->racine, objet, arbre->comparer);
}
```

Exemple : rechercher 19 dans l'arbre de la Figure 78.

3.3.2.b Insertion dans un arbre binaire ordonné

L'ajout se fait toujours au niveau d'une feuille. L'algorithme procède en 2 étapes :

- recherche dans l'arbre permettant de déterminer la feuille où doit se faire l'insertion,
- création du nœud et modification du lien père.

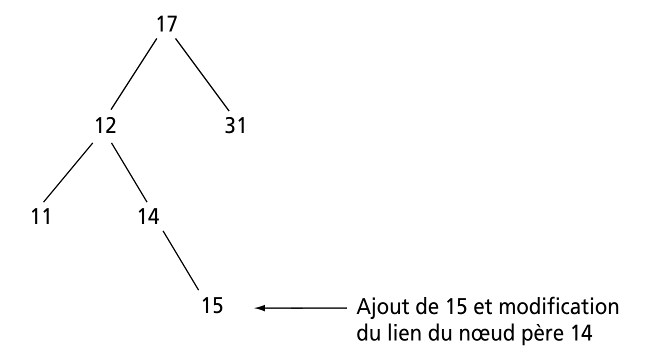


Figure 79 Ajout de 15 dans un arbre ordonné.

La structure de l'arbre dépend de l'ordre d'insertion des éléments. L'insertion suivant les ordres : (10, 20, 30), (20, 10, 30) ou (30, 20, 10) donne trois arbres ordonnés différents comme l'indique la Figure 80. Si les valeurs sont déjà ordonnées (ordre croissant ou décroissant), on aboutit à un arbre dégénéré.

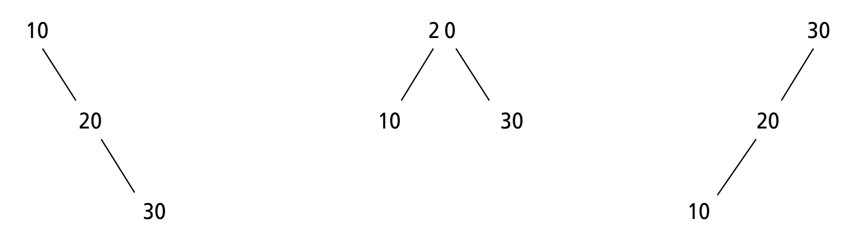


Figure 80 La forme de l'arbre dépend de l'ordre d'insertion des éléments.

La fonction d'insertion s'apparente à une fonction de recherche dans un arbre ordonné. Il faut trouver la feuille où l'insertion doit se faire. Pour cela, la comparaison de l'élément à insérer et de la clé du nœud visité permet de savoir si l'insertion doit se faire dans le SAG ou le SAD. On transmet à la fonction récursive, non pas le pointeur sur le SAG ou sur le SAD comme dans rechercherOrd(), mais l'adresse du pointeur du SAG ou du SAD, de façon à pouvoir le modifier quand on arrive sur une feuille et qu'il faut rattacher au père le nouveau nœud à créer. On veut modifier un pointeur de nœud, il faut passer en paramètre l'adresse du pointeur, soit un pointeur de pointeur de nœud. En Pascal, il faut passer le paramètre en var pour indiquer qu'il s'agit d'une adresse de pointeur. Les notations sont un peu compliquées en C.

Premier cas : insertion de 17 dans un arbre vide. pracine repère la racine de l'arbre qui est vide et vaut donc NULL. Il faut créer la feuille 17 (*fonction cF()*) et la rattacher à l'adresse contenue dans pracine.

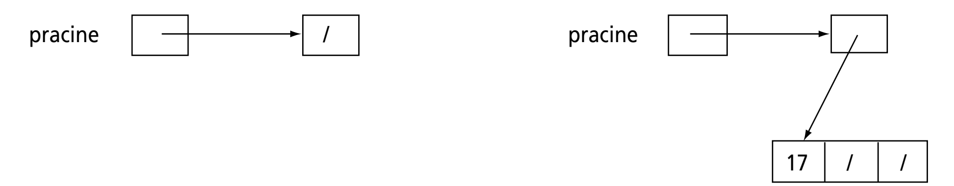


Figure 81 Insertion dans un arbre ordonné vide.

Deuxième cas : insertion de 12 dans l'arbre contenant déjà 17. 12 étant inférieur à 17, il faut insérer dans le SAG du nœud 17. On transmet à la fonction appelée récursivement l'adresse pracine du pointeur du SAG. L'utilisation de la variable locale racine permet de simplifier les notations, sinon il faudrait écrire en partant de pracine : `insererArbreOrd ( &(*pracine)->gauche, objet, comparer) ;` de même pour l'appel récursif à droite.



Figure 82 Insertion dans un arbre ordonné (cas général).

```
// pracine : pointeur sur la racine à modifier
void insererArbreOrd (Noeud** pracine, Objet* objet,
                                           int (*comparer) (Objet*, Objet*)) {
```

```
    Noeud* racine = *pracine;
    int resu;

    if (racine == NULL) {
      racine   = cF (objet);
      *pracine = racine;
    } else if ( (resu = comparer (objet, racine->reference)) == 0 ) {
      printf ("objet existe déjà dans l'arbre\n");
    } else if (resu < 0 ) {
      insererArbreOrd (&racine->gauche, objet, comparer);
    } else {
      insererArbreOrd (&racine->droite, objet, comparer);
    }
  }

  void insererArbreOrd (Arbre* arbre, Objet* objet) {
    insererArbreOrd (&arbre->racine, objet, arbre->comparer);
  }
```

3.3.2.c Minimum, maximum

Pour trouver la valeur minimum d'un arbre ordonné, il faut suivre le chemin le plus à gauche dans l'arbre.

```
// min de l'arbre ordonné
static Objet* minArbreOrd (Noeud* racine) {
  Objet* resu;
  if (racine==NULL) {
    resu = NULL;
  } else if (racine->gauche == NULL) {
    resu = racine->reference;
  } else {
    resu = minArbreOrd (racine->gauche);
  }
  return resu;
}

Objet* minArbreOrd (Arbre* arbre) {
  return minArbreOrd (arbre->racine);
}
```

Pour trouver la valeur maximum d'un arbre ordonné, il faut suivre le chemin le plus à droite dans l'arbre.

```
// max de l'arbre ordonné
Objet* maxArbreOrd (Noeud* racine) {
  Objet* resu;
  if (racine==NULL) {
    resu = NULL;
  } else if (racine->droite == NULL) {
    resu = racine->reference;
  } else {
    resu = maxArbreOrd (racine->droite);
  }
  return resu;
}

Objet* maxArbreOrd (Arbre* arbre) {
  return maxArbreOrd (arbre->racine);
}
```

3.3.2.d Suppression d'un élément dans un arbre binaire ordonné

Si le sous-arbre droit du nœud à supprimer est vide comme lors de la suppression de 28 sur la Figure 78, il suffit de modifier le pointeur du père et de le faire pointer sur le SAG du nœud à supprimer. Le SAD de 19 pointe sur le nœud 26. De même pour la suppression de 35 où le pointeur du père est remplacé par le SAG de 35 soit NULL.

Si le sous-arbre gauche du nœud à supprimer est vide comme lors de la suppression de 11 sur la Figure 78, il suffit de modifier le pointeur du père et de le faire pointer sur le SAD du nœud à supprimer. Le SAG de 17 pointe sur le nœud 15.

Sinon, dans le cas général, SAG et SAD ne sont pas NULL. La valeur à supprimer est remplacée dans le nœud par la valeur la plus grande du sous-arbre gauche. Le nœud contenant cette valeur maximum est supprimé. On pourrait également prendre la valeur la plus petite du sous-arbre droit. La suppression de 31 sur la Figure 78 se fait en remplaçant 31 par la valeur la plus grande du SAG de 31 soit 28 et en supprimant l'emplacement anciennement occupé par 28.

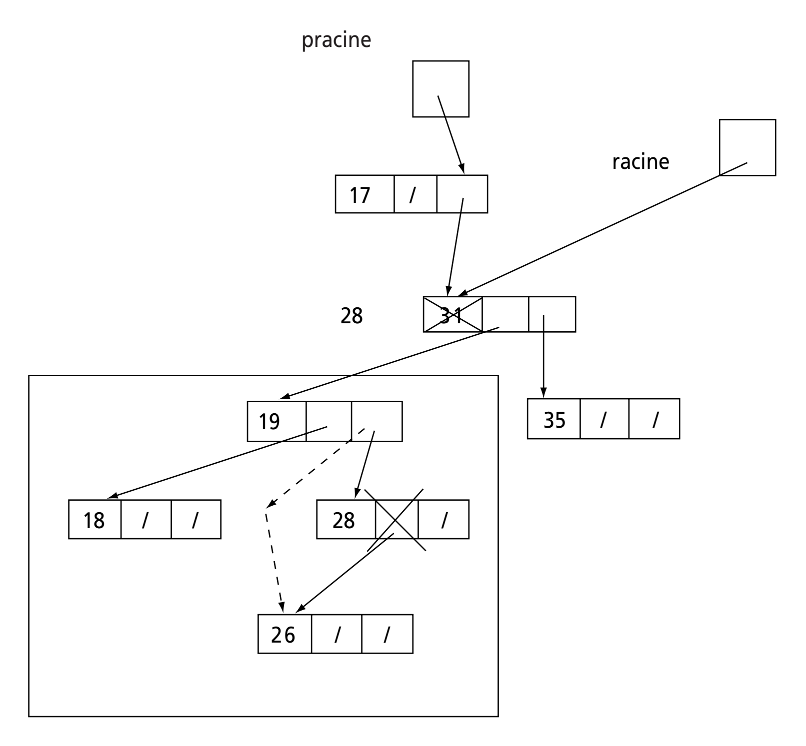


Figure 83 Suppression de 31 : exemple du cas général.

Pour trouver le plus grand dans le SAG, il faut parcourir le sous-arbre en allant toujours à droite jusqu'à ce que le SAD soit vide, ce qui est le cas du nœud 28 sur l'exemple de la Figure 83. Il faut alors faire pointer le nœud père de 28 sur le SAG de 28.

```
// appelé avec sag et sad de *racine différents de NULL;
// fournit un pointeur sur le plus grand en partant de *pracine
Noeud* supMax (Noeud** pracine) {
  Noeud* racine = *pracine;
  Noeud* pg;        // plus grand

  if (racine->droite == NULL) {   // racine repère le plus grand
    pg      = racine;
    racine  = racine->gauche;   // SAG de PG
    *pracine = racine;
  } else {
    pg = supMax (&racine->droite);
  }
  return pg;
}

Noeud* supprimerArbreOrd (Noeud** pracine, Objet* objet,
                                      int (*comparer) (Objet*, Objet*)) {
  Noeud* racine = *pracine;
  Noeud* extrait;
  int    resu;

  if (racine == NULL) {
    extrait = NULL;
  } else if ( (resu=comparer(objet, racine->reference)) < 0 ) {
    extrait = supprimerArbreOrd (&racine->gauche, objet, comparer);
  } else if (resu > 0) {
    extrait = supprimerArbreOrd (&racine->droite, objet, comparer);
  } else {
    extrait = racine;
    if (extrait->droite == NULL) {            // pas de SAD
      racine = extrait->gauche;
    } else if (extrait->gauche == NULL) {     // pas de SAG
      racine = extrait->droite;
    } else {
      extrait = supMax (&racine->gauche);
      // permuter les références des objets
      Objet* temp = racine->reference;
      racine->reference = extrait->reference;
      extrait->reference = temp;
    }
  }

  *pracine = racine;
  return extrait;
}

Noeud* supprimerArbreOrd (Arbre* arbre, Objet* objet) {
  return supprimerArbreOrd (&arbre->racine,objet, arbre->comparer);
}
```

3.3.2.e Parcours infixé droite gauche (décroissant)

Le parcours infixé droite-gauche fournit les éléments en ordre décroissant. Voir parcours d'arbres Figure 63, page 114.

```
// toString fournit la chaîne de caractères à écrire pour un objet
static void infixeDG (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine != NULL) {
    infixeDG (racine->droite, toString);
```

```
    printf ("%s ", toString (racine->reference));
    infixeDG (racine->gauche, toString);
  }
}

// parcours infixé droite gauche de l'arbre
void infixeDG (Arbre* arbre) {
  infixeDG (arbre->racine, arbre->toString);
}
```

3.3.3 Menu de test des arbres ordonnés de chaînes de caractères

Les arbres ordonnés de chaînes de caractères référencent dans chaque nœud un objet de type chaîne de caractères. Suivant le type de la clé, il faut faire une comparaison de chaînes ascii ou d'entiers. Ainsi, en terme de chaînes ascii, "11" < "9" mais en terme d'entiers 11>9. La clé est toujours mémorisée sous forme d'ascii.

```
// comparer deux chaînes de caractères
// fournit <0 si ch1 < ch2; 0 si ch1=ch2; >0 sinon
int comparerCar (Objet* objet1, Objet* objet2) {
  return strcmp ((char*)objet1, (char*)objet2);
}
```

```
// comparer des chaînes de caractères correspondant à des entiers
// 9 < 100 (mais pas en ascii)
int comparerEntierCar (Objet* objet1, Objet* objet2) {
  long a = atoi ((char*) objet1);
  long b = atoi ((char*) objet2);
  if (a==b) {
    return 0;
  } else if (a<b) {
    return -1;
  } else {
    return 1;
  }
}
```

On utilise les fonctions *infixe()* et *dessinerArbre()* du module des arbres binaires qui permettent à l'aide du menu suivant de visualiser les modifications de l'arbre après chaque ajout ou retrait dans l'arbre ordonné.

```
/* pparbreordonne.cpp  programme principal des arbres ordonnés */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "arbre.h"
#include "mdtypes.h"

// créer un arbre ordonné de chaînes de caractères à partir du fichier fe
Arbre* creerArbreOrd (FILE* fe, int (*comparer) (Objet*, Objet*)) {
  Arbre* arbre = creerArbre (NULL, toChar, comparer);
  while (!feof(fe)) {
    char* message = (char*) malloc (20);
    fscanf (fe, "%s", message);
    insererArbreOrd (arbre, message);
```

```
  }
  return arbre;
}

int menu () {
  printf ("\n\nARBRES ORDONNES\n\n");
  printf ("0 - Fin du programme\n");
  printf ("\n");
  printf ("1 - Initialisation    d'un arbre ordonné\n");
  printf ("2 - Création          à partir de fichier\n");
  printf ("3 - Parcours infixé   (croissant)\n");
  printf ("4 - Parcours infixéDG (décroissant)\n");

  printf ("\n");
  printf ("5 - Insertion         d'un élément\n");
  printf ("6 - Recherche         d'un élément\n");
  printf ("7 - Suppression       d'un élément\n");
  printf ("\n");
  printf ("8 - Dessin            à l'écran\n");
  printf ("9 - Dessin            dans un fichier\n");
  printf ("\n");
  printf ("Votre choix de 0 à 9 ? ");

  int cod; scanf ("%d", &cod); getchar();
  printf ("\n\n");
  return cod;
}

void main () {
  int  typCle;   // 1 : ascii, 2 : entier
  int (*comparer) (Objet*, Objet*);
  Arbre* arbre = creerArbre();

  booleen fini = faux;
  while (!fini) {

    switch  ( menu() ) {

    case 0 :
      fini = vrai;
      break;

    case 1 :
      detruireArbre (arbre);
      printf ("Type de la clé (1:ascii; 2:entière) ? ");
      scanf  ("%d", &typCle); getchar();
      comparer = typCle==1 ? comparerCar : comparerEntierCar;
      arbre = creerArbre (NULL, toChar, comparer);
      break;

    case 2 : {
      printf ("Type de la clé (1:ascii; 2:entière) ? ");
      scanf  ("%d", &typCle); getchar();
      comparer = typCle==1 ? comparerCar : comparerEntierCar;

      printf ("Nom du fichier contenant les valeurs ? ");
      char nomFE [50]; scanf ("%s", nomFE); getchar();
      FILE* fe = fopen (nomFE, "r");
      if (fe==NULL) {
        printf ("%s erreur ouverture\n", nomFE);
      } else {
        arbre = creerArbreOrd (fe, comparer);
        fclose (fe);
```

```
        dessinerArbre (arbre, stdout);
      }
      } break;

    case 3 :
      printf ("Parcours infixé (croissant)\n");
      infixe (arbre);
      break;

    case 4 :
      printf ("Parcours infixéDG (décroissant)\n");
      infixeDG (arbre);
      break;

    case 5 : {  // insertion
      printf ("Valeur à insérer ? ");
      char* message = (char*) malloc(20);
      scanf ("%s", message); getchar();
      Noeud* resu = rechercherOrd (arbre, message);
      if (resu != NULL ) {
        printf ("Nom %s existe déjà dans l'arbre\n", message);
      } else {
        insererArbreOrd (arbre, message);
        dessinerArbre   (arbre, stdout);
      }
      } break;

    case 6 : {  // recherche
      printf ("Nom recherché ? ");
      char nom[20]; scanf ("%s", nom); getchar();
      if ( rechercherOrd (arbre, nom) == NULL ) {
        printf ("Nom %s inconnu\n", nom);
      } else {
        printf ("Nom %s existe dans l'arbre ordonné\n", nom);
      }
      } break;

    case 7 : {
      printf ("Nom à supprimer ? ");
      char nom[20]; scanf ("%s", nom); getchar();
      Noeud* extrait;
      if ( (extrait = supprimerArbreOrd (arbre, nom)) == NULL ) {
        printf ("Nom %s inconnu\n", nom);
      } else {
        printf ("Nom %s supprimé dans l'arbre\n", nom);
        free (extrait);
        dessinerArbre (arbre, stdout);
      }
      } break;

    case 8 :
      dessinerArbre (arbre, stdout);
      break;

    case 9 : {
      printf ("Nom du fichier recevant le dessin ? ");
      char nomFS [50]; scanf  ("%s", nomFS); getchar();
      FILE* fs = fopen (nomFS, "w");
      if (fs == NULL) {
        perror ("dessin");
      } else {
        dessinerArbre (arbre, fs);
```

```
          fclose (fs);
        }
        } break;

      } // switch

      if (!fini) {
        printf ("\nTaper Return pour continuer"); getchar();
      }
    }  // while
  }
```

3.3.3.a Exemple de consultation des arbres ordonnés

L'arbre est construit à partir du fichier de valeurs *nombres.dat* contenant les clés 17 11 15 14 31 19 18 28 26 35 de la Figure 78.

```
ARBRES ORDONNES

0 - Fin du programme

1 - Initialisation    d'un arbre ordonné
2 - Création          à partir de fichier
3 - Parcours infixé   (croissant)
4 - Parcours infixéDG (décroissant)

5 - Insertion         d'un élément
6 - Recherche         d'un élément
7 - Suppression       d'un élément

8 - Dessin            à l'écran
9 - Dessin            dans un fichier

Votre choix de 0 à 9 ? 8

       |
       |
 _____17__________
 |               |
 |               |
11____      _____31__
     |      |       |
     |      |       |
   _15    _19____  35
   |      |     |
   |      |     |
  14     18   _28
              |
              |
             26

Taper Return pour continuer
```

ou encore :

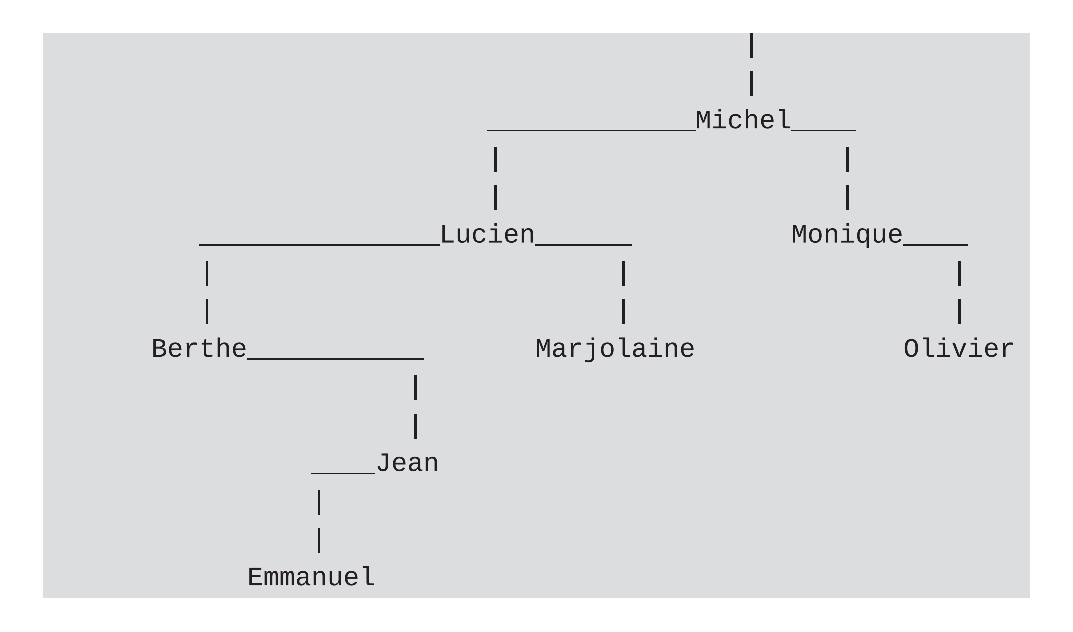
```
Choix 3 :
Parcours infixé (croissant)

11 14 15 17 18 19 26 28 31 35
```

Choix 4 :

```
Parcours infixéDG (décroissant)
35 31 28 26 19 18 17 15 14 11
```

L'insertion dans cet ordre des prénoms suivants : *Michel, Lucien, Monique, Berthe, Jean, Olivier, Marjolaine, Emmanuel*, conduit à l'arbre ordonné suivant :



```
                                     |
                                     |
                     _____________Michel____
                     |                     |
                     |                     |
   _______________Lucien______          Monique____
   |                         |                    |
   |                         |                    |
Berthe___________       Marjolaine             Olivier
                |
                |
          ____Jean
          |
          |
      Emmanuel
```

3.4 LES ARBRES BINAIRES ORDONNÉS ÉQUILIBRÉS

3.4.1 Définitions

Un arbre binaire ordonné est **parfaitement équilibré** si en tout nœud de l'arbre :

- le nombre de nœuds dans le sous-arbre gauche
- et le nombre de nœuds dans le sous-arbre droit

diffèrent au plus de 1. Ce critère est difficile à maintenir car il nécessite des réorganisations importantes, voire des reconstructions complètes de l'arbre. On utilise alors un critère d'équilibre amoindri et on parle d'arbre équilibré.

Un arbre binaire ordonné est **équilibré** (balancé ou de type AVL) si en tout nœud de l'arbre :

- la hauteur du sous-arbre gauche
- et la hauteur du sous-arbre droit

diffèrent au plus de 1. Ces arbres sont appelés arbres (binaires ordonnés) équilibrés, ou arbre AVL, initiales des personnes ayant proposé cette structure ; (AVL : introduits en 1960 par Adelson-Velskii et Landis).

La hauteur d'un nœud correspond à la longueur de la plus longue branche en partant de ce nœud (voir § 3.2.5.e, page 124). On peut démontrer mathématiquement que la hauteur totale d'un arbre équilibré est toujours inférieure à 1.5 fois la hauteur d'un arbre parfaitement équilibré ayant le même nombre de nœuds. Ajouts et suppressions peuvent **déséquilibrer** un arbre équilibré, il faut alors le réorganiser en gardant le critère d'arbre binaire ordonné équilibré. Cette structure évite d'avoir un arbre dégénéré où la recherche d'un élément est **dégradée** et devient une recherche séquentielle dans une liste.

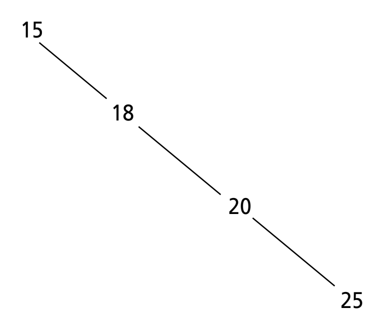


Figure 84 Arbre ordonné dégénéré.

3.4.2 Ajout dans un arbre ordonné équilibré

Une réorganisation doit avoir lieu si la valeur absolue de la différence de hauteurs entre la plus longue branche du SAD et celle du SAG d'un nœud devient > 1 à la suite d'une insertion. Il y a quatre cas de réorganisations de l'arbre à envisager : le déséquilibre vient du SAG du SAG (noté GG), du SAD du SAD (noté DD), du SAD du SAG (noté GD), du SAG du SAD (noté DG). Les deux premiers et deux derniers cas sont symétriques. Le *facteur d'équilibre* (hauteur du sous-arbre droit moins hauteur du sous-arbre gauche) est donné entre parenthèses sur les figures qui suivent.

3.4.2.a 1° cas : type GG

Le déséquilibre vient de la Gauche du sous-arbre Gauche. *a, b* sont des nœuds (*a*<*b*) ; SA1, SA2, SA3 sont des sous-arbres équilibrés. Suite à l'insertion dans SA1, le nœud *a* a un facteur d'équilibre de –1 ; le nœud *b* devrait avoir un facteur d'équi-

libre de –2, ce qui est inacceptable. Il faut réorganiser *a* et *b* comme l'indique la Figure 85.

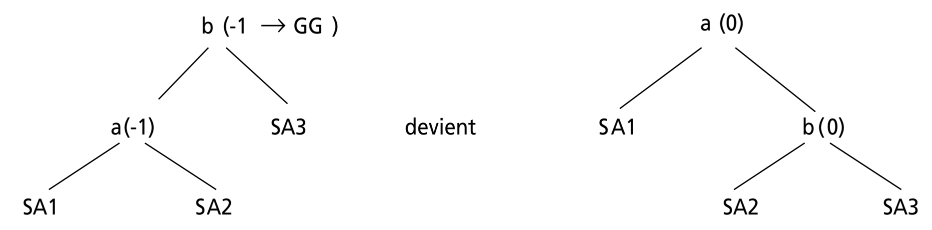


Figure 85 Principe de la réorganisation GG (Rotation Droite).

L'insertion de 1, sans réorganisation, à une place imposée par le critère d'ordre, déséquilibrerait l'arbre comme l'indique la Figure 86. La branche en gras indique le chemin suivi par les différents appels récursifs pour atteindre le point d'insertion qui est toujours une feuille. L'intérêt de la méthode réside dans le fait que seule cette branche est affectée par une éventuelle réorganisation de l'arbre pour respecter le critère d'ordre. La réorganisation se fait au retour de l'appel récursif, lors de la remontée de la feuille vers la racine de l'arbre. La feuille insérée (ici 1) est équilibrée. Le retour au nœud 3 trouve un facteur d'équilibre de 0 avant insertion qui devient –1 après insertion. On remonte au nœud 6 qui passe de même de 0 à –1. Le nœud 15 a un facteur d'équilibre de –1 avant l'insertion qui a accentué le déséquilibre à gauche.

Il faut réorganiser les nœuds 6 et 15. 6 devient racine du sous-arbre à la place de 15 qui glisse à droite de 6. Le sous-arbre 9 est rattaché comme SAG de 15. Après ces modifications, les nœuds 6 et 15 ont un facteur d'équilibre de 0. Les autres nœuds, en amont des nœuds réorganisés, examinés lors de la remontée ne sont pas touchés ; il n'y a aucun traitement à faire lors de la remontée après l'appel récursif ; sur l'exemple, le nœud 25 garde donc son facteur d'équilibre de –1. Les autres parties de l'arbre (branche droite de 25 par exemple) ne sont pas concernées par cette réorganisation.

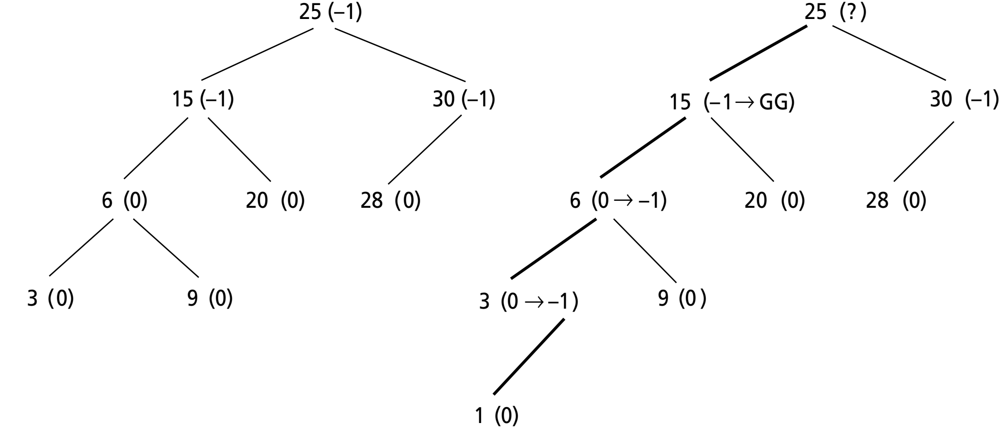


Figure 86 Arbre déséquilibré par l'insertion de 1.

L'arbre peut être réorganisé comme l'indique la Figure 87 (nœud a:6; nœud b:15), le nœud a:6 devenant la nouvelle racine du sous-arbre.

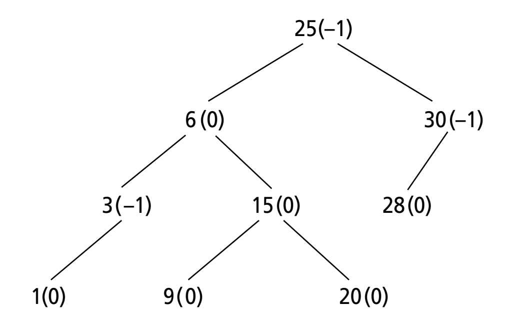


Figure 87 Arbre après rééquilibrage GG de 6 et 15.

Le détail des modifications de pointeurs sur les différents sous-arbres des nœuds concernés par une réorganisation GG est donné sur la Figure 88 et la Figure 89. Quand on exécute la fonction pour le nœud 15, ce qui a été passé lors de l'appel récursif est l'adresse pracine du pointeur sur ce nœud. Après réorganisation, le pointeur à l'adresse pracine pointe sur 6 au lieu de 15. Tout se passe comme s'il y avait eu une rotation à droite des nœuds 6 et 15 ; 6 a pris la place de 15 ; 15 descend à droite de 6 ; 9 se rattache à gauche de 15. La fonction rd (pracine) réalise cette permutation des 3 pointeurs.

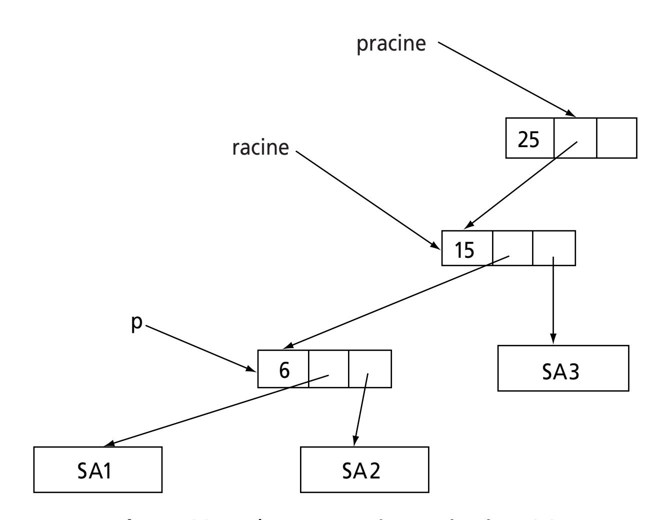


Figure 88 Arbre avant réorganisation GG.

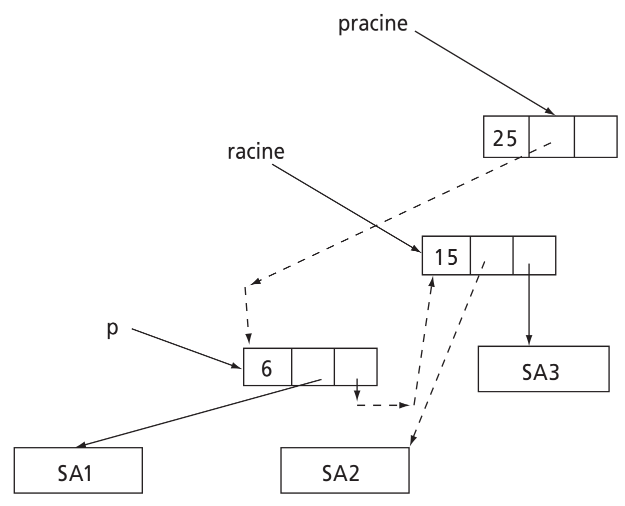


Figure 89 Arbre après réorganisation GG (Rotation Droite).

3.4.2.b 2° cas : type DD (symétrique du cas 1)

Le déséquilibre vient de la droite du sous-arbre droit. L'insertion a provoqué un déséquilibre dans le SAD du SAD. La réorganisation se fait comme indiqué sur la Figure 90, qui est symétrique de la réorganisation GG de la Figure 85.



Figure 90 Principe de la réorganisation DD (Rotation Gauche).

La Figure 91 présente un exemple simple de réorganisation DD. L'insertion de 30 déséquilibre l'arbre. Lors de la remontée, le facteur d'équilibre du nœud 20 passe de 0 à 1 ; celui de 10 qui devrait passer à 2, valeur interdite, provoque une réorganisation DD. Le principe est rigoureusement identique à celui de la réorganisation GG à la symétrie près. On parle de rotation gauche. Sur l'exemple, b:20 prend la place de 10, a:10 descend à gauche selon le principe de la Figure 90. Les facteurs d'équilibre de b:20 et a:10 sont alors de 0.

3.4.2.c 3° cas : type GD

Le déséquilibre vient de la Droite du sous-arbre Gauche (type GD). De même que précédemment a, b, c sont des nœuds (a<b<c) ; SA1, SA2, SA3, SA4 sont des sous-

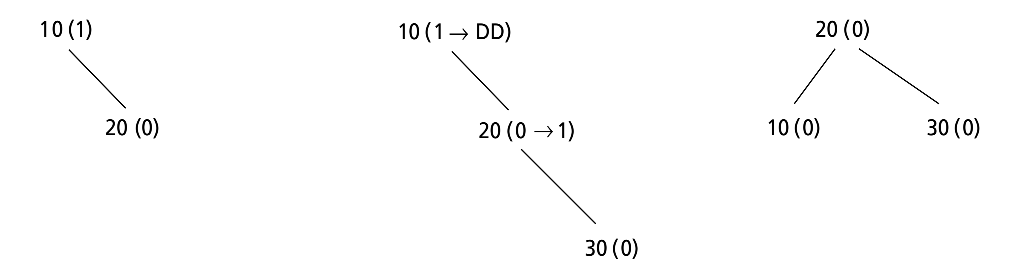


Figure 91 Exemple de réorganisation DD (Rotation Gauche).

arbres équilibrés. Le principe de la réorganisation est donné ci-dessous ; c'est b, la valeur moyenne qui devient racine du sous-arbre à la place de c.

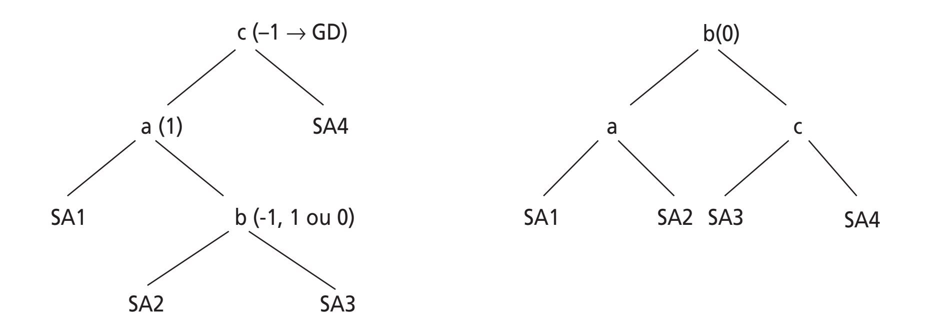


Figure 92 Principe de la réorganisation GD.

L'insertion de 8 à la place imposée par le critère d'ordre déséquilibre l'arbre comme indiqué sur la Figure 93. La branche en gras indique le chemin des appels récursifs qui a conduit à l'insertion de la feuille 8. La réorganisation se fait lors de la remontée, donc avec une séquence d'instructions qui suit l'appel récursif. La feuille 8 est équilibrée. Le nœud 9 voit son facteur d'équilibre passer de 0 à –1 (on remonte par la gauche ; l'insertion a eu lieu dans le SAG de 9). Le nœud 6 passe de 0 à 1 (on remonte par la droite, l'insertion a eu lieu dans le SAD de 6). Le facteur d'équilibre du nœud 15 (-1) devrait passer à –2 car l'insertion a eu lieu dans le SAG de 15.

Il faut donc réorganiser. Le déséquilibre vient du SAD du SAG de 15. Les nœuds c:15, a:6 et b:9 sont réorganisés conformément à la Figure 92. b:9 devient racine du sous-arbre à la place de c:15 qui glisse à droite du nœud b:9. Le nœud b:9 est devenu équilibré et les autres nœuds en amont (25 sur l'exemple) ne sont plus concernés par ce rééquilibrage. Dans le cas de l'insertion de 8, le facteur d'équilibre pour a:6 est de 0 et celui de c:15 de 1. Dans ce rééquilibrage GD, il y a trois cas à considérer du point de vue des facteurs d'équilibre.

Avant			Après		
a	b	c	a	b	c
1	-1	-1	0	0	1
1	1	-1	-1	0	0
1	0	-1	0	0	0

Avant la réorganisation sur le nœud c lors de la remontée récursive, le facteur d'équilibre de c est de –1 ; celui de a vaut +1, et celui de b vaut –1, 1 ou 0 d'où les 3 cas illustrés par le tableau et les exemples suivants.

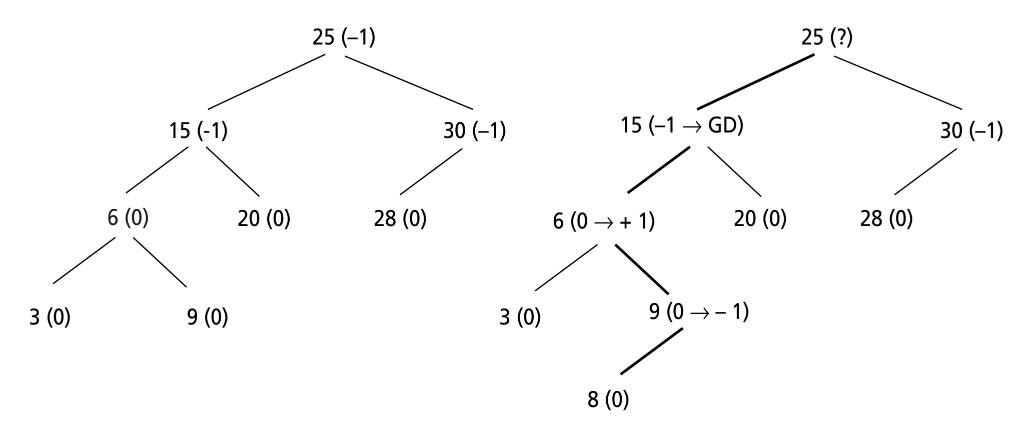


Figure 93 Arbre déséquilibré par l'insertion de 8 ; 15, 6 et 9 sont réorganisés.

L'arbre peut être réorganisé comme l'indique la Figure 94 (nœud c:15 ; nœud a:6 ; nœud b:9), le nœud b:9 compris entre a:6 et c:15 devenant la nouvelle racine du sous-arbre.

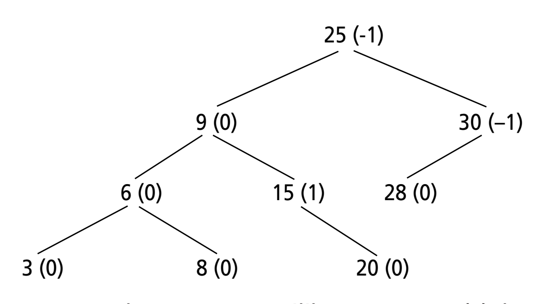


Figure 94 Cas 1 : arbre après rééquilibrage GD; a:6 (0), b:9 (0), c:15 (1).

L'insertion de 10 est schématisé sur la Figure 95 et la Figure 96 et illustre le deuxième cas concernant les facteurs d'équilibre.

La Figure 97 illustre le troisième cas concernant les facteurs d'équilibre dans un rééquilibrage GD.

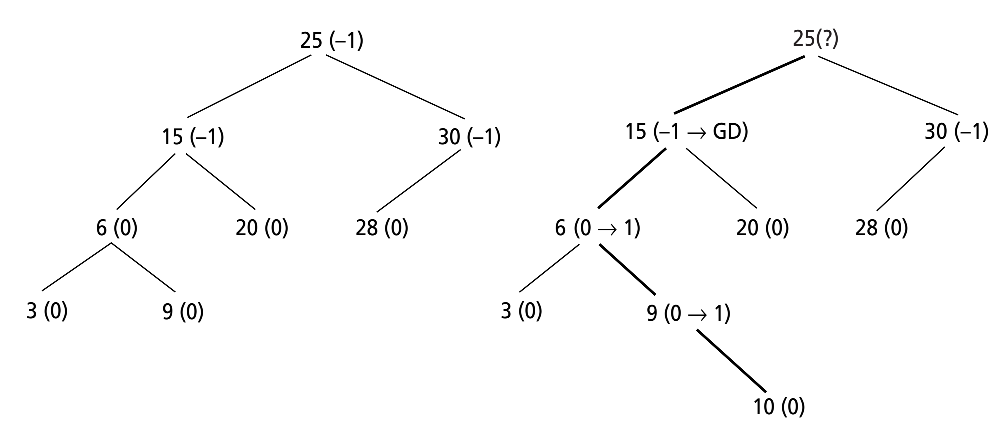


Figure 95 Arbre déséquilibré par l'insertion de 10 ; 15, 6 et 9 sont réorganisés.

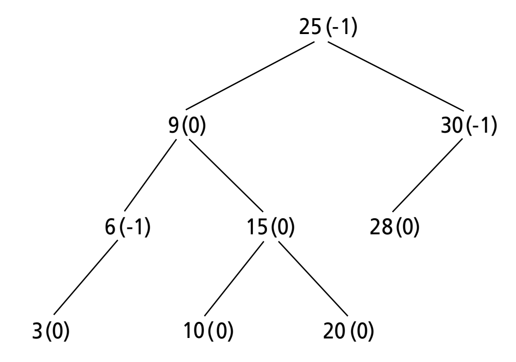


Figure 96 Cas 2 : arbre après rééquilibrage GD ; a:6 (-1), b:9 (0), c:15 (0).

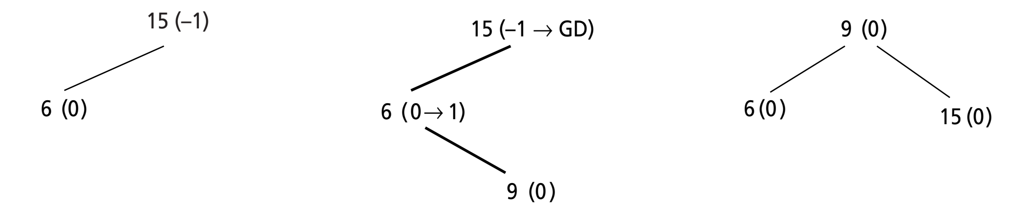


Figure 97 Cas 3 : arbre après rééquilibrage GD ; a:6 (0), b:9 (0), c:15 (0).

Du point de vue des modifications de pointeurs, cette transformation peut être considérée comme une double rotation : rotation gauche sur le nœud *a*, puis rotation droite sur le nœud *c* comme le schématise la Figure 98.

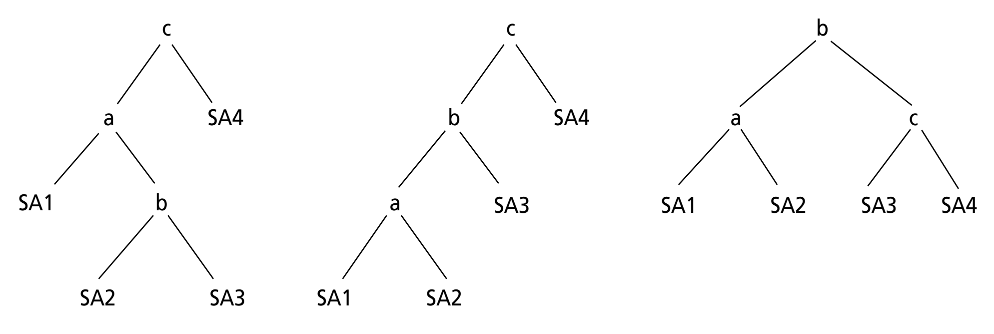


Figure 98 Réorganisation GD = rotation RG sur a, puis rotation RD sur c.

3.4.2.d 4° cas : type DG (symétrique du cas 3)

Le déséquilibre vient de la gauche du sous-arbre droit (type DG). Si le déséquilibre vient de la gauche du sous-arbre droit, il faut réorganiser comme l'indique la Figure 99. Ceci peut aussi se décomposer en une rotation droite sur c, puis une rotation gauche sur a. Il y a de même trois cas à considérer, pour les facteurs d'équilibre, symétriques des cas GD.

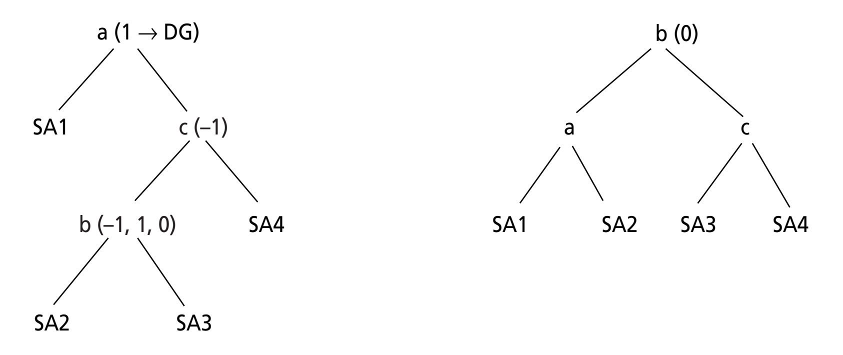


Figure 99 Principe de la réorganisation DG.

La Figure 100 présente un exemple de réorganisation DG. L'insertion de 15 se fait à gauche de 20. Le facteur d'équilibre de 20 passe de 0 à –1, celui de 10 devrait passer à 2 ; il provoque une réorganisation DG.

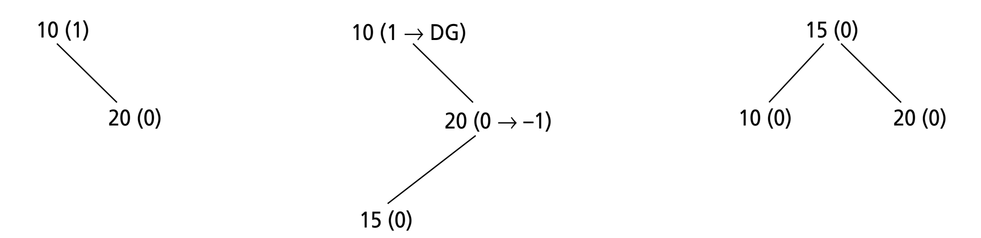


Figure 100 Exemple de réorganisation DG.

Pour les facteurs d'équilibre, on retrouve les 3 cas du type GD. Le tableau est rigoureusement le même que pour le cas GD car afin d'avoir a, b et c en ordre croissant, on a permuté c et a dans la symétrie GD/DG.

Exercice 18 - Facteur d'équilibre

Retrouver du point de vue des facteurs d'équilibre, les 3 cas de rééquilibrage DG en insérant dans un arbre vide :

50, 40, 70, 80, 60, 58
50, 40, 70, 80, 60, 65
10, 20, 15

3.4.2.e Insertion dans un arbre binaire équilibré

Les paragraphes 3.4.2.a, b, c et d ont présenté les quatre réorganisations de l'arbre. Il n'y a pas réorganisation à chaque ajout d'un élément dans l'arbre ordonné (en moyenne, une réorganisation pour 2 ajouts). Dans certains cas, l'insertion améliore l'équilibre de l'arbre comme sur la Figure 101 et la Figure 102.

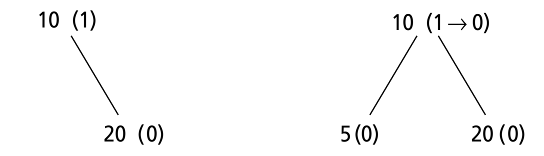


Figure 101 L'insertion de 5 dans le SAG améliore l'équilibre du nœud 10.



Figure 102 L'insertion de 30 en SAD améliore l'équilibre du nœud 20.

Les notations concernant les arbres équilibrés sont celles décrites pour les arbres binaires dans arbre.h. Toutefois, il convient d'ajouter le champ facteur d'équilibre *factEq* pour chaque nœud de l'arbre. factEq vaut -1, 0 ou 1 ; c'est la différence de hauteur entre le SAD et le SAG. Comme pour les arbres ordonnés, la clé peut être numérique ou alphanumérique.

```
// ********* ARBRES EQUILIBRES

// permuter p1, p2, p3 : p2 = p1, p3 = p2, p1 = p3
static void permut (Noeud** p1, Noeud** p2, Noeud** p3) {
  Noeud* temp;
```

```
  temp = *p3;
  *p3  = *p2;
  *p2  = *p1;
  *p1  = temp;
}

// rotation droite
static void rd (Noeud** pracine) {
  Noeud* racine = *pracine;
  Noeud* p      = racine->gauche;
  permut (pracine, &p->droite, &racine->gauche);
}

// rotation gauche
static void rg (Noeud** pracine) {
  Noeud* racine = *pracine;
  Noeud* p      = racine->droite;
  permut (pracine, &p->gauche, &racine->droite);
}


// - insérer objet dans l'arbre d'adresse de racine "pracine";
// - req à vrai indique qu'il se peut qu'un rééquilibrage
//     soit nécessaire en amont du noeud en cours
static void insererArbreEquilibre (Noeud** pracine, Objet* objet,
                                                         booleen* req,
                                            char* (toString) (Objet*),
                                 int (*comparer) (Objet*, Objet*) ) {
  int    resu;
  Noeud* racine = *pracine;

  if (racine == NULL) {
    racine         = cNd (objet);
    racine->factEq = 0;
    *req           = vrai;
    *pracine       = racine;

  } else if ( (resu=comparer (objet, racine->reference)) < 0) {
    insererArbreEquilibre (&racine->gauche, objet, req, toString, comparer);
    if (*req) {
      // L'insertion a eu lieu dans le SAG de racine
      switch (racine->factEq) {
      case 1:  // 1 -> 0
        fprintf (stderr, "%s 1 -> 0\n", toString (racine->reference));
        racine->factEq = 0;
        *req           = faux;
        break;
      case 0:  // 0 -> -1
        racine->factEq = -1;
        fprintf (stderr, "%s 0 -> -1\n", toString (racine->reference));
        break;
      case -1:
        if (racine->gauche->factEq==-1) { // cas GG
         fprintf (stderr, "%s -1 -> GG\n", toString (racine->reference));
          Noeud* b  = racine;
          Noeud* a  = b->gauche;
          b->factEq = 0;
          a->factEq = 0;
          rd (pracine);
        } else {          // cas GD
         fprintf (stderr, "%s -1 -> GD\n", toString (racine->reference));
          Noeud* c  = racine;
```

```
          Noeud* a  = c->gauche;
          Noeud* b  = a->droite;
          c->factEq = b->factEq == -1 ?  1 : 0;
          a->factEq = b->factEq ==  1 ? -1 : 0;
          b->factEq = 0;
          rg (&racine->gauche);
          rd (pracine);
        }
        *req = faux; // pas de rééquilibrage en amont de racine
        break;
      }
    }

  } else if (resu > 0) {
    insererArbreEquilibre (&racine->droite, objet, req, toString,
                                                           comparer);
    if (*req) {
      // L'insertion a eu lieu dans le SAD de racine
      switch (racine->factEq) {
      case -1:  // -1 -> 0
        fprintf (stderr, "%s -1 -> 0\n", toString (racine->reference));
        racine->factEq = 0;
        *req           = faux;
        break;
      case 0:   // 0 -> 1
        racine->factEq = 1;
        fprintf (stderr, "%s 0 -> 1\n", toString (racine->reference));
        break;
      case 1:
        if (racine->droite->factEq == 1) {   // cas DD
          fprintf (stderr, "%s 1 -> DD\n", toString (racine->reference));
          Noeud* a  = racine;
          Noeud* b  = a->droite;
          a->factEq = 0;
          b->factEq = 0;
          rg (pracine);
        } else {                // cas DG
          fprintf (stderr, "%s 1 -> DG\n", toString (racine->reference));
          Noeud* a  = racine;
          Noeud* c  = a->droite;
          Noeud* b  = c->gauche;
          c->factEq = b->factEq == -1 ?  1 : 0;
          a->factEq = b->factEq ==  1 ? -1 : 0;
          b->factEq = 0;
          rd (&racine->droite);
          rg (pracine);
        }
        *req = faux; // pas de rééquilibrage en amont
        break;
      }
    }
  } else {
    *req = faux; // déjà dans l'arbre
  }
}

void insererArbreEquilibre (Arbre* arbre, Objet* objet) {
  booleen req;
  insererArbreEquilibre (&arbre->racine, objet, &req, arbre->toString,
                                                    arbre->comparer);
}
```

3.4.3 Exemple de test pour les arbres équilibrés

On pourrait de même que pour les arbres ordonnés faire un menu permettant de tester les différentes fonctions des arbres équilibrés (voir § 3.3.3, page 163). Les fonctions de parcours *infixe()*, *infixeDG()*, et de recherche *rechercherOrd()* restent valables pour l'arbre équilibré. Pour le dessin, on peut concaténer le facteur d'équilibre au nom du nœud, la fonction *dessinerArbre()* restant valable. La fonction d'insertion a été vue précédemment. La fonction de suppression est laissée à titre d'exercice.

Exemples de test :

```
ARBRES ORDONNES EQUILIBRES
0 - Fin du programme

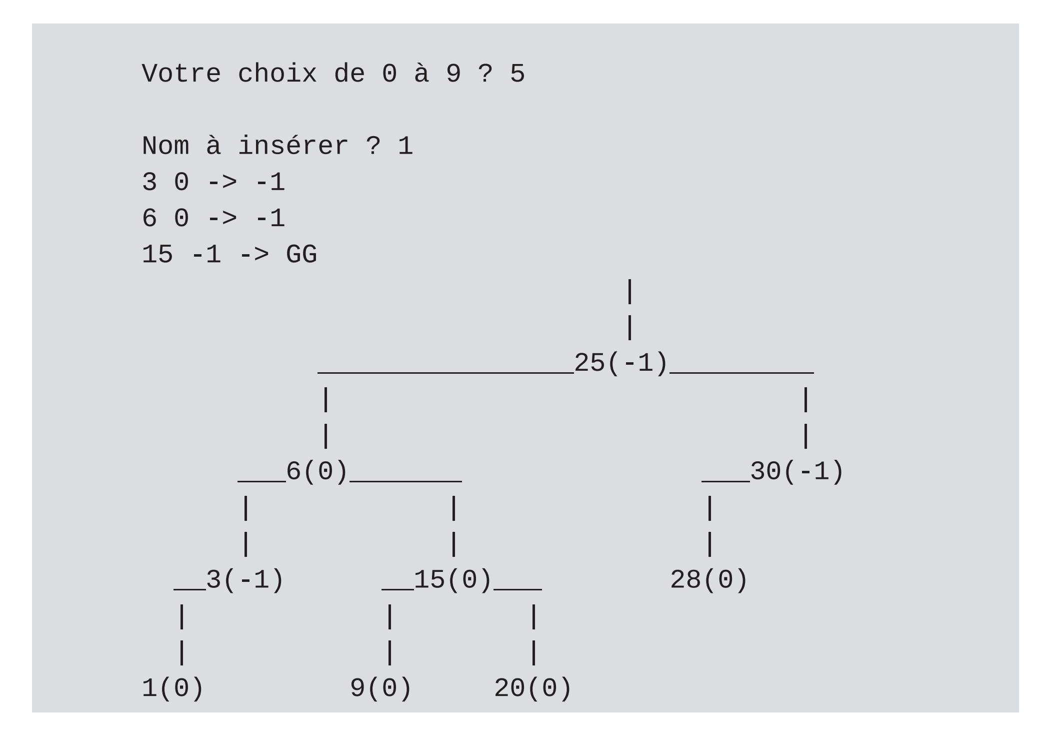
1 - Initialisation     d'un arbre ordonné   vide
2 - Création           d'un arbre équilibré (fichier)
3 - Parcours infixé    de l'arbre ordonné   (croissant)
4 - Parcours infixéDG  de l'arbre ordonné   (décroissant)


5 - Insertion          d'un élément dans l'arbre équilibré
6 - Recherche          d'un élément de   l'arbre
7 - Suppression        d'un élément de   l'arbre A FAIRE


8 - Dessin             de l'arbre courant à l'écran
9 - Dessin             de l'arbre courant dans un fichier

Votre choix de 0 à 9 ? 8

                             |
                             |
                    ________25(-1)_________
                    |                     |
                    |                     |
           ______15(-1)___            ___30(-1)
           |              |           |
           |              |           |
       __6(0)___       20(0)       28(0)
       |       |
       |       |
     3(0)    9(0)
```



```
Votre choix de 0 à 9 ? 5

Nom à insérer ? 1
3 0 -> -1
6 0 -> -1
15 -1 -> GG
                              |
                              |
              ________________25(-1)_________
              |                             |
              |                             |
         ___6(0)_______                 ___30(-1)
         |             |                |
         |             |                |
     __3(-1)       __15(0)___         28(0)
     |             |        |
     |             |        |
   1(0)          9(0)     20(0)
```

L'arbre est d'abord construit à partir de clés contenues dans un fichier, puis affiché sur le premier écran. L'insertion de 1 sur l'écran 2 ci-dessus correspond à la réorganisation de la Figure 86 et de la Figure 87. La trace des modifications des facteurs d'équilibre des nœuds lors de la remontée récursive est également indiquée pour les nœuds 3, 6 et 15.

Les instructions suivantes effectuent la construction et le dessin d'un arbre équilibré de Personne. La fonction *dupliquerArbreBal()* dessine l'arbre en indiquant le facteur d'équilibre.

```
// dupliquer l'arbre en insérant le facteur d'équilibre
// dans le message de l'objet
Noeud* dupliquerArbreBal (Noeud* racine, char* (*toString) (Objet*)) {
  if (racine==NULL) {
    return NULL;
  } else {
    char* message = (char*) malloc(30);
    sprintf (message, "%s(%d)", toString (racine->reference),
                                                   racine->factEq);
    Noeud* nouveau  = cF (message);
    nouveau->gauche = dupliquerArbreBal (racine->gauche, toString);
    nouveau->droite = dupliquerArbreBal (racine->droite, toString);
    return nouveau;
  }
}

// dupliquer un arbre en insérant dans chaque noeud,
// la chaîne de caractères du noeud et le facteur d'équilibre
```

```
Arbre* dupliquerArbreBal (Arbre* arbre) {
  Noeud* nracine = dupliquerArbreBal (arbre->racine, arbre->toString);
  return creerArbre (nracine, toChar, comparerCar);
}

void main () {
  Personne* p1 = creerPersonne ("Dupont", "Jacques");
  Personne* p2 = creerPersonne ("Dupond", "Albert");
  Personne* p3 = creerPersonne ("Dufour", "Aline");
  Personne* p4 = creerPersonne ("Dupré",  "Berthe");
  Personne* p5 = creerPersonne ("Duval",  "Sébastien");

  // arbre de Personne
  Arbre* arbrep = creerArbre (NULL, toStringPersonne, comparerPersonne);
  insererArbreEquilibre (arbrep, p1);
  insererArbreEquilibre (arbrep, p2);
  insererArbreEquilibre (arbrep, p3);
  insererArbreEquilibre (arbrep, p4);
  insererArbreEquilibre (arbrep, p5);

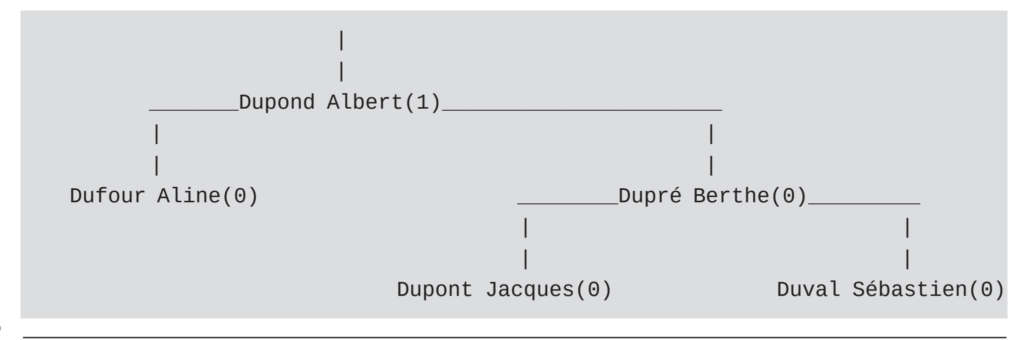
  printf ("\nordre croissant :\n");
  infixe (arbrep);

  printf ("\nordre décroissant :\n");
  infixeDG (arbrep);
  printf ("\n");

  printf ("dessin de l'arbre équilibré de personnes\n");
  dessinerArbre (arbrep, stdout);

  // dessin de l'arbre auquel on ajoute le facteur d'équilibre
  Arbre* narbrep = dupliquerArbreBal (arbrep);
  printf ("dessin de l'arbre équilibré de personnes
                                          et du facteur d'équilibre\n");
  dessinerArbre (narbrep, stdout);
}
```

L'arbre équilibré et les facteurs d'équilibre pour chaque nœud :



```
                    |
                    |
       _______Dupond Albert(1)______________________
       |                                           |
       |                                           |
Dufour Aline(0)                        ________Dupré Berthe(0)_________
                                       |                              |
                                       |                              |
                             Dupont Jacques(0)             Duval Sébastien(0)
```

Exercice 19 - Insertion de valeurs dans un arbre équilibré

Donner les différents schémas de réorganisation concernant l'insertion des nombres entiers de 1 à 12 dans un arbre équilibré. Faire de même en partant de 12 vers 1.

3.5 ARBRES N-AIRES ORDONNÉS ÉQUILIBRÉS : LES B-ARBRES

3.5.1 Définitions et exemples

Les B-arbres sont des arbres n-aires ordonnés conçus pour de grands ensembles de données sur mémoire secondaire. Chaque nœud contient plusieurs valeurs ordonnées qui délimitent les ensembles de valeurs que l'on peut trouver dans les sous-arbres. Pour un B-arbre d'ordre N il y a, pour tout nœud sauf la racine, de N à 2*N valeurs et donc au plus 2*N + 1 intervalles ou sous-arbres. La racine contient de 1 à 2*N valeurs. Un nœud est rebaptisé « page » dans la terminologie B-arbre. Les valeurs courantes de N sur disque vont de 25 à 200. Chaque nœud a un nombre minimum et un nombre maximum de valeurs ; chaque accès disque permet de lire une page en mémoire centrale dans un tableau contenant les différentes valeurs et leurs caractéristiques. La hauteur de l'arbre est faible, donc le nombre d'accès disque pour retrouver une valeur est faible. On réserve de la place pour 2*N valeurs dans chaque nœud mais un nœud contient de N à 2*N valeurs. Il y a donc allocation inutilisée d'espace mémoire. On retrouve le compromis espace-temps. On accélère la recherche au détriment de l'espace mémoire.

Exemple de principe de B-arbre d'ordre 2 (de 2 à 4 valeurs par nœud, de 3 à 5 fils)

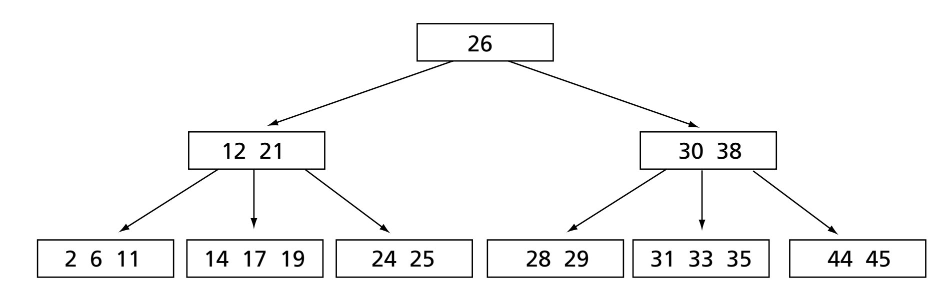


Figure 103 B-arbre d'ordre 2.

Description d'un B-arbre d'ordre N

```
#define N    2
typedef char ch15 [16];
typedef int  PNoeud;

typedef  struct {
  ch15   cle;
  int    numEnr; // le numéro d'enregistrement des informations
  PNoeud p;
} Element;

typedef struct {
  int     nbE;   // nombre réels d'éléments dans le noeud
  PNoeud  p0;
  Element elt [2*N];
} Noeud;
```

2 12 20

2 / 10 / 11 /

2 / 13 / 15 /

4 / 21 / 23 / 25 / 26 /

Figure 104 Détail de la structure d'un nœud d'un B-arbre d'ordre 2.

Les informations spécifiques de l'application ne sont pas indiquées sur les différents schémas concernant les B-arbres. Une possibilité simple est de créer un fichier d'index sous forme d'un B-arbre et un fichier de données en accès direct. Ainsi, sur la Figure 105, la clé 20 fait référence à l'enregistrement numEnr=3 soit le troisième enregistrement (*20 Dufour, etc.*) du fichier en accès direct Clients. La recherche d'un enregistrement se fait donc par consultation de l'index, puis lecture directe de l'enregistrement du fichier de données. Par la suite, les données ne sont pas mentionnées car elles varient d'une application à l'autre.

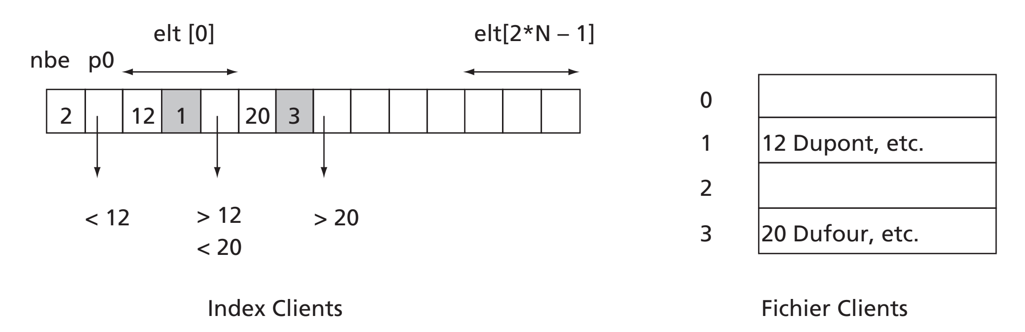


Figure 105 Fichier index (B-arbre) et fichier de données.

Toutes les feuilles sont au même niveau. Les accès disque étant relativement lents, cette structure permet de diminuer le nombre d'accès disque pour retrouver une valeur. Un B-arbre d'ordre N est saturé si tous ses nœuds contiennent exactement 2*N valeurs. Il est dit squelettique si tous ses nœuds sauf la racine contiennent exactement N valeurs. On peut utiliser une recherche dichotomique pour retrouver, en mémoire centrale, la clé ou le sous-arbre contenant la clé parmi les nbE clés du nœud.

3.5.2 Insertion dans un B-arbre

La contrainte *nombre de valeurs compris entre N et 2*N* doit être respectée pour tout nœud (toute page) sauf pour la racine.

Algorithme d'insertion

L'insertion se fait toujours au niveau d'une feuille. Soit v la valeur à insérer et racine la racine du B-arbre au départ.

si racine est une feuille alors // insérer la valeur v dans le nœud

s'il reste de la place dans racine,
– insérer la valeur v à sa place dans racine

sinon // éclatement du nœud racine en 2 nœuds : racine et nouveau
– créer un nouveau nœud nouveau
– déterminer la clé médiane des clés du nœud (v inclus)
– placer les valeurs supérieures à la clé médiane dans la nouvelle page nouveau, celles inférieures restant dans racine,
– insérer la clé médiane dans le nœud père (le créer s'il n'existe pas) qui à son tour peut éclater, et ce jusqu'à la racine, seul moyen d'augmenter la hauteur de l'arbre. Ceci est fait lors de la remontée dans l'arbre, au retour de l'appel récursif, de façon semblable à la réorganisation dans les arbres ordonnés équilibrés.

sinon //on continue le parcours de l'arbre n-aire
– rechercher le sous-arbre concerné par v et recommencer (récursivité) avec pour racine, la racine du sous-arbre
finsi

Pour insérer une valeur, on descend toujours jusqu'au niveau d'une feuille avant de remonter éventuellement pour insérer la clé médiane. La seule façon d'augmenter la taille de l'arbre se fait par éclatement de nœuds contenant déjà 2*N valeurs. L'arbre est toujours **équilibré** quel que soit l'ordre des valeurs insérées. Cependant, la structure de l'arbre dépend de l'ordre des valeurs entrées. Le taux de remplissage peut varier de 50 % à 100 % suivant que tous les nœuds contiennent N ou 2*N valeurs. L'ajout de valeurs déjà en ordre croissant conduit à un arbre squelettique.

Exemples d'insertion de valeurs dans un B-arbre d'ordre 2

Exemple 1

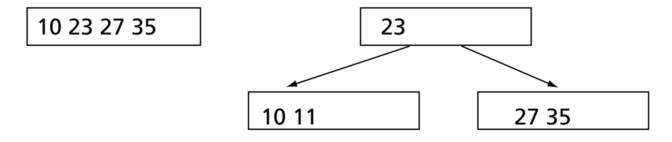


Figure 106 Insertion de 11 dans un B-arbre (avant et après).

Exemple 2

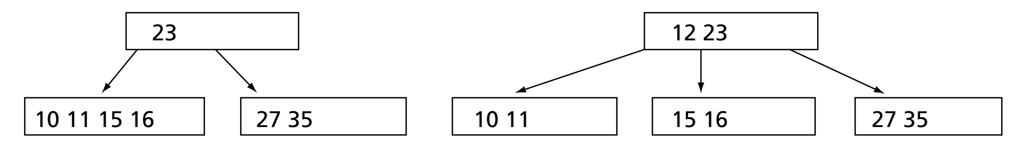
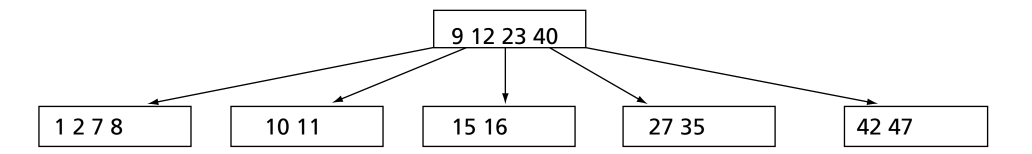


Figure 107 Insertion de 12 dans un B-arbre ; éclatement du nœud 10.11.15.16.

Exemple 3



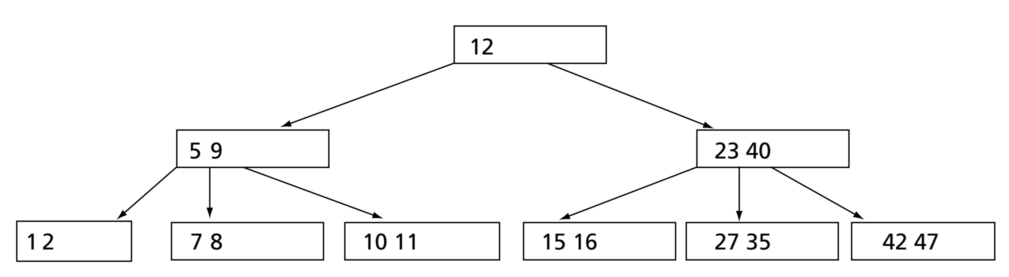


Figure 108 Insertion de 5 ; éclatement de 1.2.7.8, puis de 9.12.23.40.

3.5.3 Recherche, parcours, destruction

3.5.3.a Recherche d'un élément : accès direct

Il s'agit d'accéder directement à un élément du B-arbre. Le parcours de l'arbre n-aire ordonné permet de retrouver facilement une valeur par descente récursive dans un des sous-arbres. La valeur peut se trouver dans un nœud intermédiaire ou dans une feuille. Si on atteint une feuille sans trouver la valeur cherchée c'est que la valeur n'est pas dans l'arbre.

3.5.3.b Parcours des éléments : accès séquentiel

Il s'agit d'énumérer toutes les valeurs du B-arbre. Le parcours ressemble au parcours de l'arbre binaire, mais cette fois, il y a (sauf pour la racine) de N+1 à 2N+1 sous-arbres qu'il faut parcourir récursivement.

```
void parcoursBArbre (PNoeud racine) {
  if (racine != NULL) {
    parcoursBArbre (racine->p0);
    for (int i=0; i<racine->nbE; i++) {
      printf ("%s ", racine->elt [i].cle);
      parcoursBArbre (racine->elt [i].p);
    }
  }
}
```

3.5.3.c Destruction d'un élément

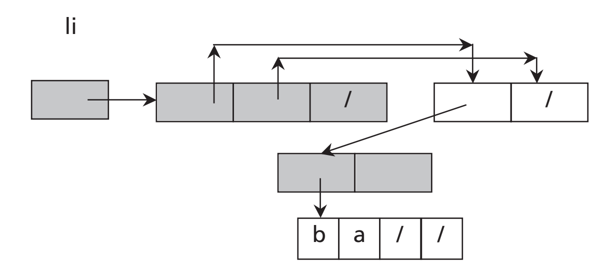
La destruction d'un élément est a priori simple. Il faut localiser le nœud où se trouve la clé et l'enlever. Cependant, si la clé est dans un nœud **non feuille**, le retrait de la clé va poser des problèmes pour accéder aux valeurs du sous-arbre qui avaient été rangées en fonction de cette valeur disparue. Il faut la remplacer par soit la plus grande valeur du sous-arbre à gauche de la valeur retirée, soit par la plus petite du sous-arbre à droite (voir § 3.3.2.d, page 161 : suppression dans un arbre ordonné). Si pour un nœud le nombre de valeurs devient **inférieur à N** et si une des pages voisines (gauche ou droite) a plus de N valeurs, il y a redistribution entre ces 2 pages, sinon les 2 nœuds (le nœud courant et celui de droite par exemple) sont regroupés et un des nœuds est détruit. La valeur médiane est recalculée.

Exercice 20 - Gestion d'un tournoi à l'aide d'une liste d'arbres

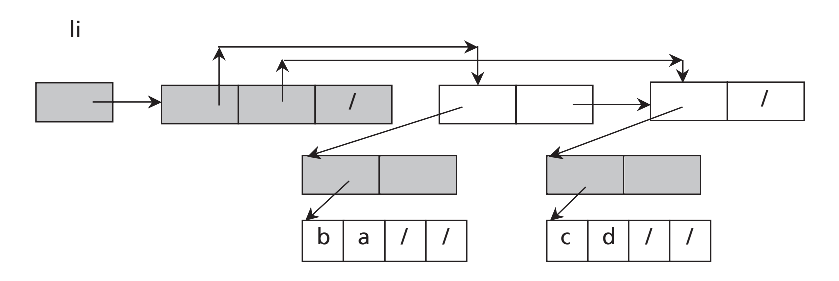
(voir § 3.1.2.e, page 105)

On souhaite développer un logiciel qui permette d'enregistrer et de consulter les résultats d'un tournoi. À partir de ces résultats enregistrés dans un fichier séquentiel, on construit une liste d'arbres mémorisant les différents matchs joués. À la fin du tournoi, la liste ne contient plus qu'un seul élément qui pointe sur la racine de l'arbre des matchs. Cette racine contient des informations sur le match de finale. On désire également répondre à des interrogations sur les résultats des matchs déjà enregistrés. La construction de la liste d'arbres est schématisée sur les exemples suivants :

- un seul match joué, le fichier séquentiel contient :
  b,a b a gagné contre a



- après enregistrement de :
  b,a b a gagné contre a
  c,d c a gagné contre d



• après enregistrement de : b,a ; c,d ; b,c ; h,g ; f,e ; h,f ;

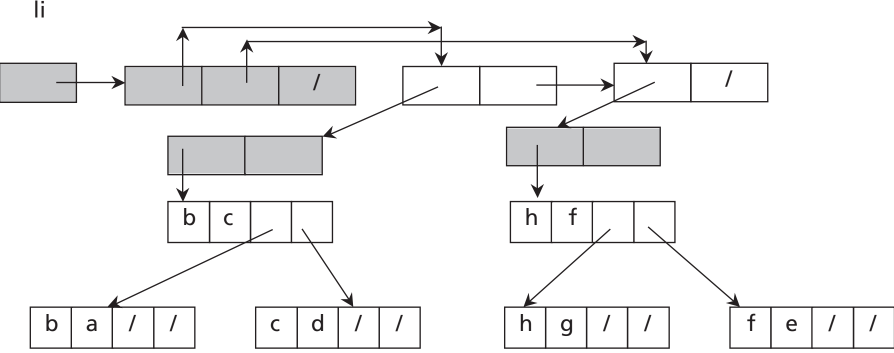


Figure 109 Une liste d'arbres.

• après enregistrement de tous les résultats du tournoi b,a ; c,d ; b,c ; h,g ; f,e ; h,f ; b,h, il ne reste plus qu'un seul arbre dans la liste.

Les schémas précédents n'indiquent pas les détails de l'implémentation d'un noeud de l'arbre. Le schéma de droite de la Figure 110 indique l'implémentation réelle du noeud schématisé à gauche.



Figure 110 Détail de l'implémentation d'un nœud.

Soient les déclarations suivantes :

```
#include "liste.h"
#include "arbre.h"

#define LGMAX 20
typedef char Chaine [LGMAX+1];  // 0 de fin
typedef struct {
  Chaine joueur1;    // gagnant = joueur1
  Chaine joueur2;
} Match;
```

Écrire les fonctions suivantes de **consultation** de l'arbre :

• *void listerRestants (Liste* li) ;* qui liste les joueurs de la liste li non encore éliminés.

• *void listerArbres (Liste* li, int type) ;* qui effectue le récapitulatif des matchs en listant le contenu de chacun des arbres de la liste li. Le parcours est préfixé si type=1, postfixé si type=2. Un dessin des arbres est donné si le type=3. Les résultats attendus des parcours préfixé et postfixé de l'arbre de l'exemple sont donnés ci-dessous.

• *void listerMatch (Noeud* pNom, Chaine joueur) ;* qui à partir d'un pointeur pNom sur le dernier match enregistré pour un joueur donné, fournit la liste des matchs que ce joueur a disputés

• *void listerMatch (Liste* li, Chaine joueur) ;* qui recherche joueur dans la liste li des arbres, et fournit la liste des matchs de joueur.

Écrire les fonctions suivantes de **création** de l'arbre :

• *void enregistrerMatch (Liste* li, Chaine j1, Chaine j2) ;* qui enregistre le match gagné par j1 contre j2 dans la liste li des arbres.

• *void creerArbres (FILE* fe, Liste* li) ;* qui crée la liste li des arbres à partir du fichier séquentiel fe. Cette fonction utilise *enregistrerMatch()*.

Faire un programme correspondant au menu suivant :

```
GESTION D'UN TOURNOI

0 - Fin
1 - Création de la liste d'arbres à partir d'un fichier
2 - Enregistrement d'un match
3 - Liste des joueurs non éliminés
4 - Parcours préfixé  des arbres
5 - Parcours postfixé des arbres
6 - Dessins des arbres des matchs
7 - Liste des matchs d'un joueur
```

Exemples de résultats (choix 4) :

```
PARCOURS DE L'ARBRE

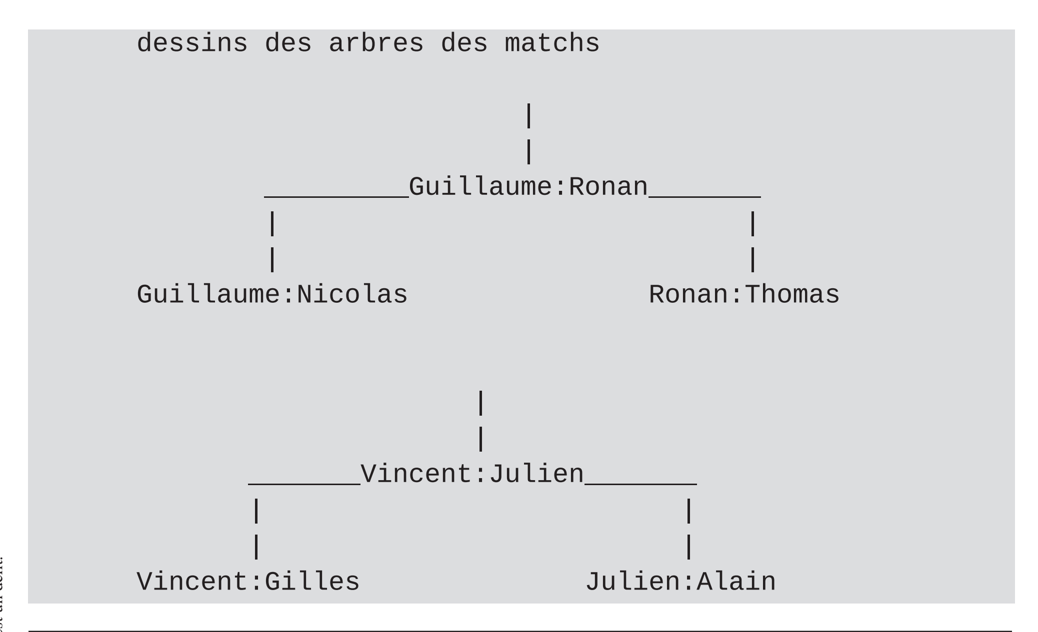
préfixé indenté :
 b gagne contre h
      b gagne contre c
           b gagne contre a
           c gagne contre d
      h gagne contre f
           h gagne contre g
           f gagne contre e
```

Exemples de résultats (choix 5) :

```
postfixé indenté :
           b gagne contre a
           c gagne contre d
      b gagne contre c
           h gagne contre g
           f gagne contre e
      h gagne contre f
 b gagne contre h


matchs du joueur e
f gagne contre e
```

Un exemple du dessin d'une liste de deux arbres de jeu (choix 6).



```
dessins des arbres des matchs

                        |
                        |
        _________Guillaume:Ronan_______
        |                             |
        |                             |
Guillaume:Nicolas               Ronan:Thomas


                     |
                     |
       _______Vincent:Julien_______
       |                          |
       |                          |
Vincent:Gilles              Julien:Alain
```

Exercice 21 - Mémorisation d'un dessin sous forme d'un arbre binaire

(allocation contiguë, voir Figure 75, page 150)

Une image peut être représentée par un tableau à deux dimensions de booléens si l'image est en noir et blanc. Une autre méthode plus économique en mémoire dans certains cas (tracés continus) est de ne représenter que les points significatifs, sous forme d'un arbre, et de restituer l'image à partir de cet arbre. On peut de plus très facilement faire varier la taille du dessin. Sur la Figure 111, le premier dessin représente un caractère 1 manuscrit. Le deuxième dessin schématise ce caractère sous

forme d'un arbre n-aire. Il faut tracer le segment 0, puis en partant de la fin du segment 0, il faut tracer le segment 1 et le segment 2. De même, en partant de la fin du segment 2, il faut tracer les segments 3, 4 et 5. Le troisième dessin représente l'arbre n-aire converti sous sa forme binaire équivalente. Les segments sont décrits avec un code indiquant la direction de déplacement (1:nord, 2:nord-est, 3:est, 4:sud-est, 5:sud, 6:sud-ouest, 7:ouest, 8:nord-ouest).

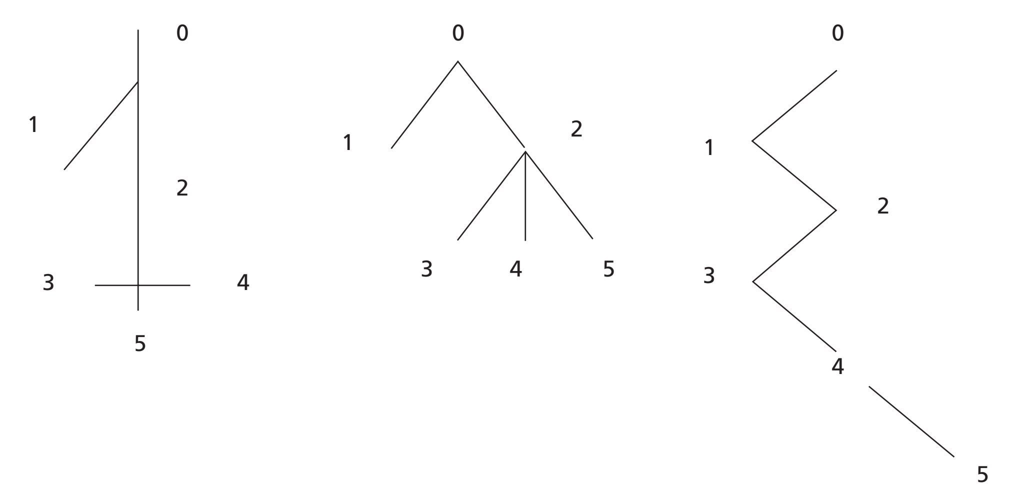


Figure 111 Mémorisation d'un tracé graphique sous forme d'arbre.

La description des différents segments est enregistrée dans le tableau *desc*. L'arbre binaire est mémorisé en allocation contiguë dans le tableau *arbre*. Les structures de données utilisées sont les suivantes :

```
#include "ecran.h"

#define NULLE -1
typedef int   PNoeud;

typedef struct {
  int    indice;   // indice sur desc []
  PNoeud gauche;
  PNoeud droite;
} Noeud;

int desc [] = {    // description de l'image
// 0                                            15
   3, 5, 5, 5, 3, 6, 6, 6, 7, 5, 5, 5, 5, 5, 5, 5,
   3, 7, 7, 7, 3, 3, 3, 3, 2, 5, 5
};

Noeud arbre [] = { // l'arbre représentant l'image
  { 0,   1, -1 },  // 0
  { 4,  -1,  2 },  // 1
  { 8,   3, -1 },  // 2
  { 16, -1,  4 },  // 3
  { 20, -1,  5 },  // 4
  { 24, -1, -1 }   // 5
};
```

indice repère le rang dans le tableau *desc* [] de la description du segment que le nœud représente. Le premier nœud de l'arbre en *arbre*[0] à un *indice* de 0 qui pointe sur l'entrée 0 du tableau *desc* [] soit 3/5/5/5. Le 3 indique le nombre de valeurs à prendre en compte ; 5 indique la direction de déplacement pour reconstituer le segment 0 soit 3 caractères '*' vers le sud.

En utilisant le module *ecran* (voir § 1.4.2, page 25) :

• écrire la fonction void traiterNoeud (PNoeud pNd, int x, int y, int taille, int* nx, int* ny) ; qui trace sur l'écran à l'aide de '*' le dessin du segment correspondant au Nœud de rang pNd de arbre[]. x et y sont les coordonnées du début du segment sur l'écran, nx et ny les coordonnées de la fin du segment. taille indique que chaque élément du tableau desc[] doit être répété taille fois.

• écrire la fonction *récursive* void parcours (PNoeud racine, int x, int y, int taille) ; qui effectue à partir de x et y, le dessin représenté par l'arbre commençant en racine :

Le programme principal est le suivant :

```
void main () {
  PNoeud racine = 0;  // premier noeud de Arbre

  initialiserEcran (40, 40);
  parcours         (racine, 10, 10, 2);
  afficherEcran    () ;
  sauverEcran      ("GrdEcran.res");
  detruireEcran    ();
}
```

et le dessin :

```
        *
        *
        *
        *
       **
      * *
        *
        *
        *
        *
      *****
        *
```

Le dessin est très simple pour faciliter les explications. On peut aussi tracer une signature, des caractères chinois ou la Loire et ses affluents.

Exercice 22 - Références croisées (arbre ordonné)

On veut déterminer les références croisées d'un texte (ou d'un programme par exemple). Ceci consiste à répertorier tous les mots du texte, à les classer par ordre alphabétique, et à imprimer une table précisant pour chaque mot, les numéros des lignes où on a rencontré le mot. Pour cela, on constitue un arbre ordonné de Nœud, chaque nœud contenant un mot et la liste des numéros de ligne où ce mot a été trouvé.

Les structures de données utilisées sont les suivantes (on utilise le module *liste.h* voir § 2.3.8.a, page 49) :

```
// les objets Elt (liste de numéros de lignes)

typedef struct {
  int numLigne;
} Elt;

// les objets Mot

typedef char Chaine [31];  // 30 + 0 de fin

// un mot et sa liste de lignes
typedef struct {
  Chaine mot;
  Liste  li;
} Mot;
```

La Figure 112 indique que *petit* a été trouvé à la ligne 1 et à la ligne 2.

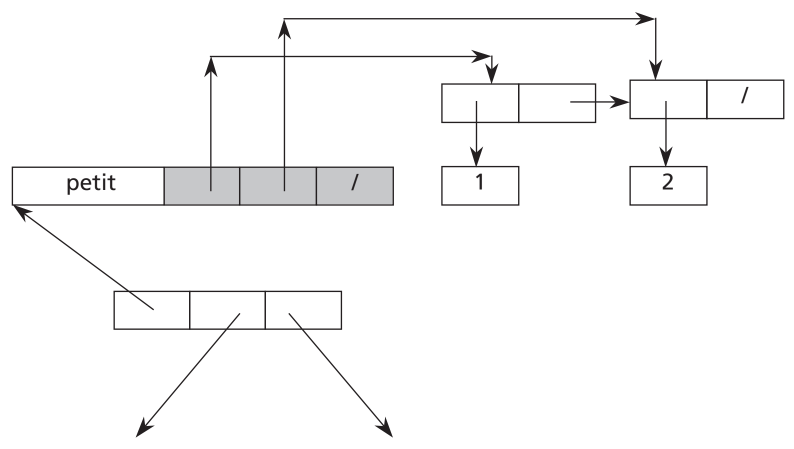


Figure 112 Un nœud de l'arbre contenant une liste.

- Écrire une fonction *void traiterNd (Mot* pmot) ;* qui écrit le mot se trouvant dans pmot suivi des numéros des lignes où ce mot a été rencontré. (Écrire 10 numéros par ligne).

• Écrire une fonction *void rechercherLignes (Arbre* arbre, char* mot) ;* qui recherche le mot `mot` de l'arbre binaire ordonné et écrit les numéros des lignes où ce mot a été trouvé.

• Écrire une fonction *void insererMot (Arbre* arbre, Chaine mot, int nl) ;* qui insère le mot mot dans l'arbre ordonné arbre. nl contient le numéro de ligne où on a trouvé mot.

• Écrire le programme « références croisées » qui à partir d'un fichier d'entrée constitue un fichier de sortie des références croisées de ce fichier. Le programme analyse les caractères et ne retient que ceux pouvant constituer des mots qu'il insère dans l'arbre ordonné. Les lignes lues sont réécrites dans le fichier de sortie, précédées de leur numéro de ligne pour vérification des références croisées. Faire le dessin de l'arbre ordonné.

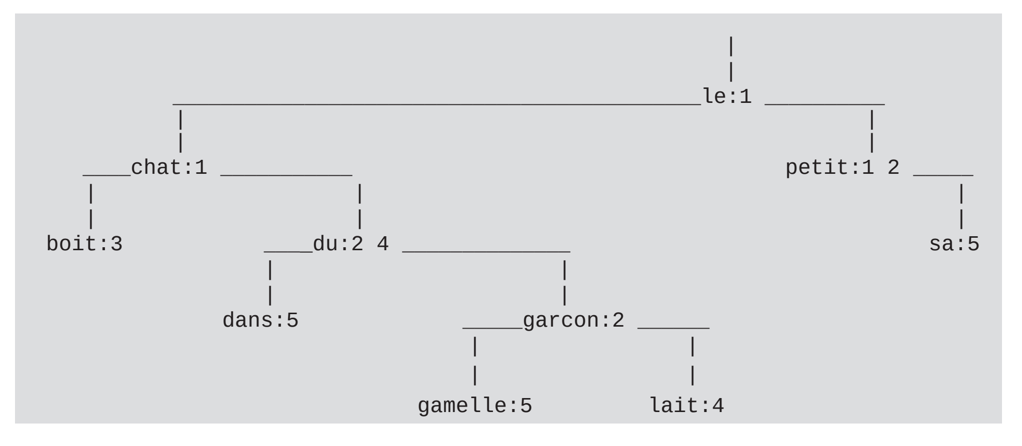
• Modifier le programme d'insertion pour que l'arbre soit équilibré. En faire le dessin.

Exemple de résultats :

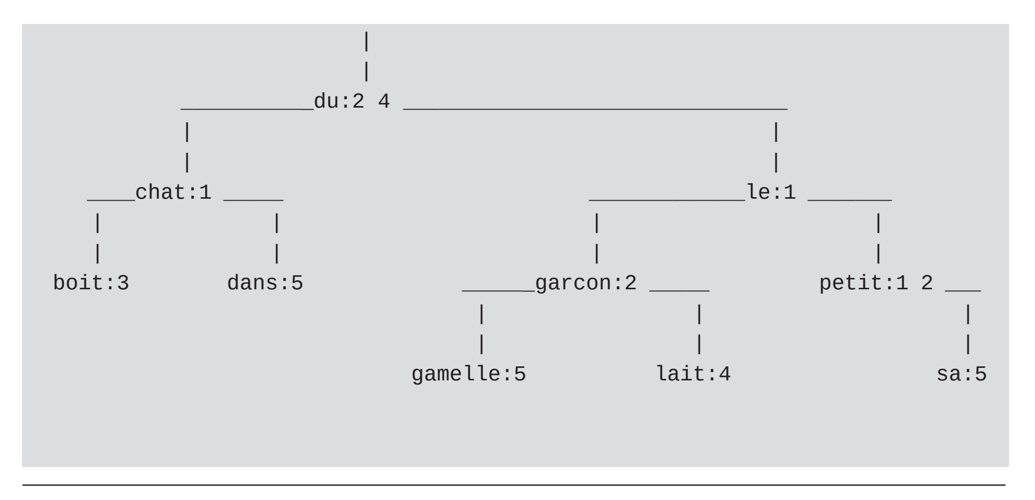
```
          1 le petit chat
          2 du petit garcon
          3 boit
          4 du lait
          5 dans sa gamelle

     Références croisées
                boit :      3
                chat :      1
                dans :      5
                  du :      2     4
             gamelle :      5
              garcon :      2
                lait :      4
                  le :      1
               petit :      1     2
                  sa :      5
```

Arbre ordonné :



Arbre équilibré :



```
                         |
                         |
           __________du:2 4 ______________________________
           |                                             |
           |                                             |
    ____chat:1 _____                       ____________le:1 _______
    |             |                        |                      |
    |             |                        |                      |
boit:3        dans:5             ______garcon:2 _____        petit:1 2 ___
                                 |                |                      |
                                 |                |                      |
                             gamelle:5         lait:4                  sa:5
```

Exercice 23 - Arbre de questions

Soit un arbre binaire de questions contenant pour chaque nœud :

- un pointeur vers le texte correspondant à la question à poser,
- un pointeur sur le sous-arbre gauche des questions,
- un pointeur sur le sous-arbre droit des questions.

Les questions sont booléennes : on peut répondre par oui ou par non. La racine de l'arbre de la Figure 113 correspond à la question « *Est-ce un homme ?* ». Le schéma ne mentionne qu'un mot, il devrait en fait contenir toute la question. Le sous-arbre gauche correspond aux questions à poser si la réponse est oui, le sous-arbre droit celles à poser si la réponse est non.

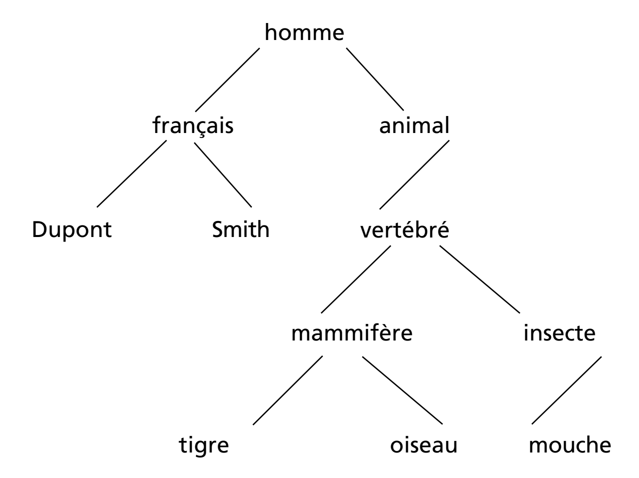


Figure 113 Arbre de questions.

• Écrire la fonction `void poserQuestions (Arbre* arbre) ;` qui permet de poser les questions d'une branche de l'arbre en fonction des réponses (O ou N) de l'utilisateur. *Exemple :*

```
Est-ce un homme ? O
Est-ce un français ? N
Est-ce Smith ? O
Fin des questions : vous avez trouvé

Est-ce un homme ? N
Est-ce un animal ? N
Fin des questions : je donne ma langue au chat
```

• Écrire une fonction qui liste les questions de l'arbre de manière indentée. *Exemple :*

```
Arbre des questions

Est-ce un homme
     Est-ce un francais
          Est-ce Dupont
          Est-ce Smith
     Est-ce un animal
          Est-ce un vertébré
               Est-ce un mammifère
                    Est-ce un tigre
                    Est-ce un oiseau
               Est-ce un insecte
                    Est-ce une mouche
```

• Écrire la fonction `void insererQuestion (Arbre* arbre) ;` qui permet d'insérer une question à réponse booléenne dans l'arbre. La nouvelle question est insérée au niveau d'une feuille après avoir répondu aux questions qui mènent à cette feuille.

Exemple :

```
Insertion d'une question

Est-ce un homme (O/N) ? n
Est-ce un animal (O/N) ? n
Question à insérer ? : Est-ce un objet
```

• Écrire la fonction `void sauverQuestions (Arbre* arbre, FILE* fs);` qui enregistre l'arbre dans le fichier fs en faisant un parcours préfixé de l'arbre, les sous-arbres nuls étant notés `"*"`.

- Écrire la fonction `void chargerQuestions (Arbre* arbre, FILE* fe);` qui crée l'arbre à partir du fichier fe créé par `sauverQuestions()`.
- Écrire le programme principal correspondant au menu suivant :

```
ARBRE DE QUESTIONS

 0 - Fin
 1 - Insérer  une nouvelle question
 2 - Lister   l'arbre des questions
 3 - Poser    les questions
 4 - Sauver   l'arbre des questions dans un fichier
 5 - Charger  l'arbre des questions à partir d'un fichier
```

3.6 RÉSUMÉ

Les arbres sont des structures de données très importantes en informatique permettant de résoudre certains problèmes de manière concise et élégante grâce aux techniques de la récursivité qui se justifie pleinement sur ces structures. Les arbres n-aires ont une structure imposée par l'application elle-même (exemple : l'arbre des continents et des pays de la Terre). Si le nombre de sous-composants (sous-arbres) est variable, il faut convertir l'arbre n-aire en un arbre binaire, tout en répondant à des questions de l'arbre n-aire.

S'il y a un critère d'ordre, on peut créer un arbre ordonné en insérant les clés les plus petites à gauche, et les plus grandes à droite. Cependant certaines branches risquent dans certaines configurations de croître démesurément. Les arbres ordonnés équilibrés évitent cet inconvénient en réorganisant l'arbre tout en gardant le critère d'ordre. Si l'arbre est sur mémoire secondaire, il vaut mieux éviter les accès disques qui sont relativement lents et regrouper plusieurs clés dans un même nœud. On retrouve un arbre n-aire ordonné appelé B-arbre. La recherche de la clé dans l'arbre ordonné permet de retrouver les informations attachées à la clé qui sont spécifiques de chaque application.

Chapitre **4**

Les tables

4.1 CAS GÉNÉRAL

4.1.1 Définition

Une table est une structure de données telle que l'accès à un élément est déterminé à partir de sa clé. La clé permet d'identifier un élément de manière unique ; la clé (un entier ou une chaîne alphanumérique) est dite discriminante. Dans la suite de ce chapitre, on n'envisage pas les cas où une même clé peut correspondre à plusieurs éléments. Connaissant la clé, on peut retrouver l'élément et ses caractéristiques. L'allocation d'espace pour une table est souvent contiguë. Il faut réserver l'espace mémoire en début de traitement (en mémoire centrale ou sur mémoire secondaire).

	Clé	Infos
0		
1		
2	Dupond	
3		
NMax -1		

Figure 114 Une table en mémoire centrale ou secondaire (fichier).

La partie Infos contient les informations spécifiques de l'application pour une clé donnée. Ainsi, pour *Dupond*, la partie Infos peut contenir (Michel, rue des mimosas par exemple). Sur mémoire secondaire, on peut aussi constituer une table d'index en mettant dans la partie Infos, un numéro d'enregistrement (25 par exemple) contenant les données qui sont elles mémorisées dans un autre fichier en accès direct. On a un fichier d'index et un fichier de données (voir Figure 115). Le fichier d'index peut être amené tout ou en partie en mémoire centrale, ce qui accélère le traitement. Sur la Figure 115, on a créé une seconde table d'index basée sur le numéro de téléphone.

	Fichier d'index sur Nom	
	Nom	**Numéro**
2	Dupont	25

	Fichier d'index sur Téléphone	
	Téléphone	**Numéro**
5	0234872222	25

	Fichier de données			
	Nom	**Prénom**	**Adresse**	**Téléphone**
25	Dupont	Michel	rue des mimosas	0234872222

Figure 115 Deux tables d'index pour un fichier de personnes.

4.1.2 Exemples d'utilisation de tables

4.1.2.a Gestion d'articles

Connaissant le numéro d'un article (HV32 par exemple), on peut retrouver la quantité en stock (200) et le prix unitaire (50.00) par consultation de la table de la Figure 116. (clé : Numéro ; Infos : QT et PU).

	Numéro	QT	PU
0			
1			
2			
3	HV32	200	50.00
4			
5			
6			

Figure 116 Une table d'articles.

4.1.2.b Table d'étiquettes dans un compilateur

Un compilateur se constitue une table à partir des différents identificateurs déclarés qu'il rencontre lors de la compilation d'un programme. Lors de la référence à un identificateur, le compilateur doit retrouver les attributs de cet identificateur. Le nom de l'identificateur sert de clé ; à partir de cette clé, on peut retrouver : son type, son adresse mémoire par rapport au début des données, etc. (clé : Nom ; Infos : Type, Adresse, etc.). Sur la Figure 117, la variable A est de type entier (1), et a pour adresse 25 (25e octet) par rapport au début des données du programme. Il y a 2 phases :

- rangement des identificateurs lors de leur déclaration,
- recherche de leurs caractéristiques lors des références.

	Nom	Type	Adresse
0	A	1	25
1			
2	C	2	10
3			
4	Fin	3	53
5			
6			

Figure 117 Une table d'un compilateur.

4.1.2.c Dictionnaire

Connaissant un mot français, on peut retrouver son équivalent anglais par consultation d'une table. (clé : mot français ; Infos : mot anglais).

4.1.2.d Remarques

Un vecteur (ou un tableau) est un cas particulier de table où la clé n'est pas mémorisée car les clés sont contiguës de 0 à n-1, n étant le nombre d'éléments dans la table.

4.1.3 Création, initialisation de tables

En mémoire centrale, la table peut se déclarer comme indiqué ci-dessous de façon à être la plus générique possible. Un objet de la table est constitué de la clé et des informations concernant cette clé. Le type Table mémorise la longueur maximum nMax de la table, le nombre n d'éléments dans la table et un pointeur vers le début de la table proprement dite allouée dynamiquement lors de l'initialisation de la table.

```
typedef void Objet;

typedef struct {
  int     nMax;     // nombre max. d'éléments dans la table
  int     n;        // nombre réel d'éléments dans la table
  Objet** element;  // un tableau de pointeur vers les objets
  char*   (*toString) (Objet*);
  int     (*comparer) (Objet*, Objet*);
} Table;
```

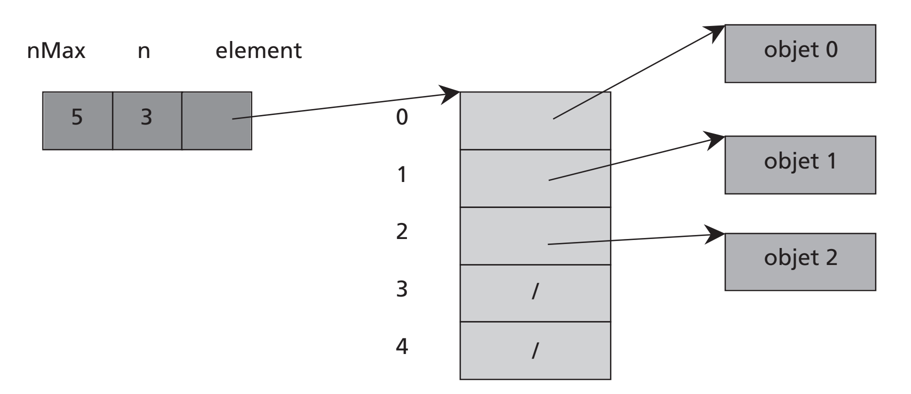


Figure 118 Le type Table (les pointeurs des objets sont consécutifs).

La fonction *creerTable()* effectue l'allocation dynamique de la table et de la partie contiguë de la table. nMax indique le nombre maximum d'éléments dans la table. La fonction *toString()* fournit une chaîne de caractères correspondant à l'objet de la table. La fonction *comparer()* compare deux objets et fournit une valeur <0, =0 ou >0 suivant que le premier objet est <, = ou > au deuxième. Voir pointeurs de fonctions, § 1.5, page 33.

```
Table* creerTable (int nMax, char* (*toString) (Objet*),
                                int (*comparer) (Objet*, Objet*)) {
  Table* table    = new Table();
  table->nMax     = nMax;
  table->n        = 0;
  table->element  = (Objet**) malloc (sizeof(Objet*) * nMax);
  table->toString = toString;
  table->comparer = comparer;
  return table;
}
```

```
// par défaut, les objets de la table sont des chaînes de caractères
Table* creerTable (int nMax) {
   return creerTable (nMax, toChar, comparerCar);voir § 4.6.1.a, page 206
}
```

La fonction *detruireTable()* désalloue la table allouée par *creerTable()*.

```
void detruireTable (Table* table) {
  free (table->element);
  free (table);
}
```

La fonction *insererDsTable()* insère l'objet pointé par nouveau en fin de la table. L'objet nouveau est créé et rempli avant l'appel de cette fonction.

```
// insérer nouveau dans la table
booleen insererDsTable (Table* table, Objet* nouveau) {
  if (table->n < table->nMax) {
    table->element [table->n++] = nouveau;
    return vrai;
  } else {
    return faux;
  }
}
```

La fonction *lgTable()* fournit le nombre d'éléments dans la table.

```
// nombre d'éléments dans la table
int lgTable (Table* table) {
  return table->n;
}
```

La fonction *fournirElement()* fournit un pointeur sur le nième objet de la table.

```
// fournir un pointeur sur le nième élément de la table
Objet* fournirElement (Table* table, int n) {
  if ( (n>=0) && (n<table->n) ) {
    return table->element [n];
  } else {
    return NULL;
  }
}
```

4.1.4 Accès séquentiel

À partir de la clé, il faut retrouver les caractéristiques de l'objet ayant cette clé. La méthode la plus simple consiste à chercher séquentiellement dans la table, *tant qu'on n'a pas atteint la fin de la table et tant qu'on n'a pas trouvé.*

4.1.4.a Accès séquentiel standard

La fonction *accesSequentiel()* utilise un booléen *trouve* pour indiquer s'il y a égalité entre les clés de ce qu'on cherche *objetCherche*, et de ce qu'il y a à la ième entrée de la table. La fonction pourrait être écrite différemment ; le codage privilégie la clarté

de l'algorithme plutôt que la concision. Si l'objet existe, la fonction fournit un pointeur sur l'objet correspondant dans la table, sinon elle retourne NULL. La fonction *comparer()* fournit 0 si les deux clés comparées sont égales.

```
// fournir un pointeur sur objetCherche,
// ou NULL si l'objet est absent
Objet* accesSequentiel (Table* table, Objet* objetCherche) {
  int     i      = 0;
  booleen trouve = faux;

  while ( (i < table->n) && !trouve) {
    trouve = table->comparer (objetCherche, table->element[i]) == 0;
    if (!trouve) i++;
  }
  return trouve ? table->element[i] : NULL;
}
```

4.1.4.b Accès séquentiel avec sentinelle

On recopie (le pointeur de) l'objet cherché dans l'élément n de la table (la première entrée libre de la table). Il est alors inutile de tester si i < table->n dans la boucle puisqu'on est sûr de trouver l'élément dans la table. Si on ne le retrouve pas avant l'entrée n, c'est que l'élément n'est pas dans la table. Les éléments sont numérotés de 0 à n-1. Une place doit être gardée libre en fin du tableau lors de l'insertion des éléments. On peut aussi dans ce cas allouer nMax+1 éléments lors du *malloc()* de *creerTable()*.

```
// méthode de la sentinelle : fournir un pointeur sur objetCherche,
// ou NULL si l'objet est absent
Objet* accesSentinelle (Table* table,  Objet* objetCherche) {
  int     i      = 0;
  booleen trouve = faux;

  table->element [table->n] = objetCherche;  // il doit rester une place
  while (!trouve) {
    trouve = table->comparer (objetCherche, table->element[i]) == 0;
    if (!trouve) i++;
  }
  return i < table->n ? table->element[i] : NULL;
}
```

4.1.4.c Évaluation de l'accès séquentiel

n étant le nombre d'éléments dans la table, il faut en moyenne **(n+1)/2** accès à la table pour retrouver un élément de la table. **n** accès sont nécessaires si l'élément n'est pas dans la table (élément inconnu).

Justification

Pour accéder au 1er élément :	1 accès	
Pour accéder au 2e élément :	2 accès	
Pour accéder au n^{e} élément :	n accès	
Pour accéder une fois aux n éléments :	1 + 2 ...+ n accès	soit : n(n+1)/2

donc : **(n+1) / 2** accès à la table en moyenne pour retrouver **un** élément de la table (si les probabilités d'accès aux n éléments sont équiréparties).

On peut placer en tête de la table les éléments qui sont recherchés le plus souvent. Il suffit de définir un compteur et de faire progresser en tête de la table les éléments le plus souvent référencés. La recherche séquentielle dans une table en mémoire centrale est suffisante si on a un nombre d'éléments inférieur à une trentaine d'éléments. Au-delà, l'accès à un élément peut devenir long. Si la table est ordonnée, les recherches infructueuses peuvent s'arrêter avant la fin de la consultation de la table.

4.1.5 Accès dichotomique (recherche binaire)

4.1.5.a Principe

Pour appliquer cette méthode de recherche dichotomique, la table doit être **ordonnée** suivant les clés. La recherche s'apparente à une recherche dans un dictionnaire qu'on ouvrirait en son milieu. Si le mot cherché est sur une des deux pages ouvertes, on a trouvé, sinon si le mot est inférieur à celui du haut des pages présentées, il faut chercher dans la première moitié du dictionnaire, sinon dans la seconde moitié. On recommence la division en deux parties égales avec la moitié choisie.

Au début, le premier élément de la table est nommé gauche, le dernier droite. On calcule l'élément du milieu = (gauche + droite)/2. Si milieu contient l'élément cherché, on a trouvé, sinon, si l'élément cherché est inférieur à celui de milieu, il faut recommencer la recherche avec la sous-table gauche:milieu-1 ; sinon l'élément cherché est supérieur à celui du milieu, il faut chercher dans milieu+1:droite. Cette fonction peut facilement s'écrire de manière récursive.

Si on recherche *Duf* dans la table de la Figure 119, l'élément du milieu est en 7. *Duf* est inférieur à *Jea*, il faut recommencer la recherche dans la sous-table 0:6. L'élément du milieu de la sous-table est alors 3 qui contient l'élément cherché et donc ses caractéristiques.

		Clé	Infos
Gauche	**0**	Bou	Bouchard
	1	Cab	Cabon
	2	Cos	Cosson
	3	Duf	Dufour
	4	Dup	Dupond
	5	Duv	Duval
	6	Gau	Gautier
Milieu	**7**	Jea	Jean
	8	Leg	Legoff
	9	Pet	Petit
	10	Rab	Raboutin
	11	Rob	Robert
	12	Tan	Tanguy
	13	Xav	Xavier
Droite	**14**	Zaz	Zazou

Figure 119 Recherche dichotomique dans une table.

Évaluations

Il faut **au maximum $\log_2 n$ accès à la table** pour trouver un élément dans une table de n éléments. Si $n = 16 = 2^4$, il faut $\log_2 16$, soit 4 accès maximum ; si $n = 1024 = 2^{10}$, il faut $\log_2 1024$, soit 10 accès maximum. D'une manière générale, si $n=2^p$ est la longueur de la table, il faut au maximum p accès à la table pour retrouver un élément.

La consultation dichotomique est **conseillée** si les phases de construction et de consultation de la table sont séparées :

- 1re phase : construction de la table par insertion et tri,
- 2e phase : consultation de la table par accès dichotomique.

Si insertion et recherche dans la table ne se font pas en deux phases distinctes, la méthode perd de l'intérêt car il faut retrier après chaque insertion.

4.1.5.b Dichotomie : version récursive

```
// dichotomie récursive
static Objet* dichotomie (Table* table, Objet* objetCherche,
                                                    int gauche, int droite) {
  Objet* resu;
  if (gauche <= droite) {
    int milieu = (gauche+droite) / 2;
    //printf ("%d %d %d\n", gauche, milieu, droite);
    int c  = table->comparer (objetCherche, table->element[milieu]);
    if (c == 0) {
      resu = table->element [milieu];
    } else if ( c < 0 ) {
      resu = dichotomie (table, objetCherche, gauche, milieu-1);
    } else {
      resu = dichotomie (table, objetCherche, milieu+1, droite);
    }
  } else {
    resu = NULL;
  }
  return resu;
}

// appel de la fonction récursive
Objet* dichotomie (Table* table, Objet* objetCherche) {
  return dichotomie (table, objetCherche, 0, table->n-1);
}
```

4.1.5.c Dichotomie : version itérative

L'algorithme récursif ci-dessus n'exécute qu'un seul appel récursif avec la première ou la seconde sous-table. On n'a jamais besoin de revenir en arrière pour explorer l'autre sous-table. La récursivité peut être remplacée par une itération.

```
// fournir un pointeur sur objetCherche
// ou NULL si l'objet est absent
Objet* dichotomieIter (Table* table, Objet* objetCherche) {
  Objet*  resu   = NULL;   // défaut
  int     gauche = 0;
  int     droite = table->n-1;
  booleen trouve = faux;
```

```
    while ( (gauche <= droite) && !trouve ) {
      int milieu = (gauche+droite) / 2;
      int c  = table->comparer (objetCherche, table->element[milieu]);
      if (c == 0) {
        resu = table->element [milieu];
        trouve = vrai;
      } else if (c < 0) {
        droite = milieu-1;
      } else {
        gauche = milieu+1;
      }
    }
    return resu;
  }
```

4.1.5.d Tri de la table

La méthode de tri bulle permet de trier la table en déplaçant seulement les pointeurs des objets de la table (voir Figure 118). La fonction *comparer()* (définie lors de *creerTable()*) permet de trier les objets quelle que soit l'application. Le tri bulle range au ième tour, en position i, le plus petit des restants non triés de i+1 à n.

```
// permuter les pointeurs des éléments n1 et n2
static void permuter (Table* table, int n1, int n2) {
  Objet* temp = table->element[n1];
  table->element[n1] = table->element[n2];
  table->element[n2] = temp;
}

// tri bulle
void trierTable (Table* table) {
  int n = lgTable(table);
  for (int i=0; i<n-1; i++) {
    for (int j=n-1; j>i; j--) {
      Objet* objet1 = fournirElement (table, j-1);
      Objet* objet2 = fournirElement (table, j);
      if (table->comparer (objet1, objet2) > 0) {
        permuter (table, j-1, j);
      }
    }
  }
}
```

4.1.5.e Listage de la table

On peut lister le contenu de la table pour vérification en utilisant la fonction *toString()* définie lors de *creerTable()*.

```
void listerTable (Table* table) {
  for (int i=0; i<lgTable(table); i++) {
    Objet* objet = fournirElement (table, i);  // ième élément
    printf ("%2d %s\n", i, table->toString (objet));
  }
}
```

4.1.6 Le module des tables

4.1.6.a Le type Table

Le fichier d'en-tête *table.h* du module des tables est le suivant :

```
/* table.h  gestion des tables */

#ifndef TABLE_H
#define TABLE_H

typedef int  booleen;
#define faux 0
#define vrai 1
typedef void Objet;

typedef struct {
  int     nMax;     // nombre max. d'éléments dans la table
  int     n;        // nombre réel d'éléments dans la table
  Objet** element;  // un tableau de pointeurs vers les objets
  char*   (*toString) (Objet*);                 voir § 1.5, page 33
  int     (*comparer) (Objet*, Objet*);
} Table;

Table*  creerTable       (int nMax, char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*));
Table*  creerTable       (int nMax);
void    detruireTable    (Table* table);

booleen insererDsTable   (Table* table, Objet* nouveau);
int     lgTable          (Table* table);
Objet*  fournirElement   (Table* table, int n);

Objet*  accesSequentiel  (Table* table, Objet* objetCherche);
Objet*  accesSentinelle  (Table* table, Objet* objetCherche);
Objet*  dichotomie       (Table* table, Objet* objetCherche);
Objet*  dichotomieIter   (Table* table, Objet* objetCherche);

void    trierTable       (Table* table);
void    listerTable      (Table* table);

#endif
```

Le corps du module *table.cpp* est donné ci-dessous. Il reprend les fonctions vues précédemment.

```
// table.cpp

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "table.h"

// fournir la chaîne de caractères de objet
// fonction par défaut
static char* toChar (Objet* objet) {
  return (char*) objet;
}
```

```
// comparer deux chaînes de caractères
// fournit <0 si ch1 < ch2; 0 si ch1=ch2; >0 sinon
// fonction de comparaison par défaut
static int comparerCar (Objet* objet1, Objet* objet2) {
  return strcmp ((char*)objet1, (char*)objet2);
}
```

plus les fonctions vues précédemment : *creerTable()*, etc., *listerTable()*.

4.1.6.b Menu de test des tables

Le programme pptable suivant est un programme de test du module des tables.

```
/* pptable.cpp programme principal de test des tables */

#include <stdio.h>
#include <stdlib.h>
#include "table.h"
#include "mdtypes.h"  // type Personne              voir § 2.4.1, page 51

int menu () {
  printf ("\n\nLES TABLES\n\n");
  printf ("0 - Fin du programme\n");
  printf ("\n");
  printf ("1 - Création à partir d'un fichier\n");
  printf ("\n");
  printf ("2 - Accès séquentiel à un élément\n");
  printf ("3 - Accès séquentiel avec sentinelle\n");
  printf ("4 - Accès dichotomique récursif\n");
  printf ("5 - Accès dichotomique itératif\n");
  printf ("6 - Accès au ième élément\n");
  printf ("7 - Listage de la table\n");
  printf ("8 - Tri de la table\n");
  printf ("\n");
  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  printf ("\n");

  return cod;
}

// lire un mot ou un nom dans le fichier fe; le ranger dans chaine
void lireMot (FILE* fe, char* chaine) {
  char c;

  fscanf (fe, "%c", &c);
  // passer les blancs avant le mot
  while( ( (c==' ') || (c=='\n') ) && !feof(fe) ) {
    fscanf (fe, "%c", &c);
  }

  char* pCh = chaine;
  // enregistrer le mot dans chaine jusqu'à trouver un séparateur
  while ( (c!=' ') && (c!='\n') && !feof (fe) ) {
    *pCh++ = c;
    fscanf (fe, "%c", &c);
  }
  *pCh = 0;
  //printf ("lireMot : %s \n", chaine);
}
```

```
void ecrirePersonne (Personne* p) {                   voir § 2.4.1, page 51
  if (p!=NULL) printf ("personne : %s\n", toStringPersonne (p));
}

#if 0  // test n°1
void main () {
  #define   NMAX  20

  Personne* p1 = creerPersonne ("aaaa", "aa");
  Personne* p2 = creerPersonne ("cccc", "cc");
  Personne* p3 = creerPersonne ("bbbb", "bb");
  Personne* p4 = creerPersonne ("eeee", "ee");
  Personne* p5 = creerPersonne ("dddd", "dd");

  Table* table = creerTable (NMAX, toStringPersonne, comparerPersonne);
  insererDsTable (table, p1);
  insererDsTable (table, p2);
  insererDsTable (table, p3);
  insererDsTable (table, p4);
  insererDsTable (table, p5);

  printf ("listerTable\n");
  listerTable (table);

  // la personne cherchée de clé "cccc"
  Personne* cherche = creerPersonne ("cccc", "?");
  Personne* trouve;
  trouve  = (Personne*) accesSequentiel (table, cherche);
  ecrirePersonne (trouve);

  trouve  = (Personne*) accesSentinelle (table, cherche);
  ecrirePersonne (trouve);

  trierTable (table);
  //printf ("listerTable triée\n");
  //listerTable (table);
  trouve  = (Personne*) dichotomieRec (table, cherche);
  ecrirePersonne (trouve);

  trouve  = (Personne*) dichotomieIter (table, cherche);
  ecrirePersonne (trouve);
}

#else  // test n°2

// lire le nom (la clé) d'une personne
Personne* lireNom (Table* table) {
  Personne* cherche = new Personne();
  printf ("Clé (nom) de la personne ? ");
  scanf ("%s", cherche->nom); getchar();
  return cherche;
}

void main () {
  #define NMAX  20
  Table* table = creerTable (NMAX, toStringPersonne, comparerPersonne);
  Personne* trouve = NULL;
  int choix;

  booleen fini = faux;
```

```
while (!fini) {

  switch (choix = menu()) {

  case 0 :
    fini = vrai;
    break;

  case 1 : { // On pourrait lire le nom du fichier
    FILE* fe = fopen ("personnes.dat", "r");
    if (fe==NULL) {
      perror ("Ouverture");
    } else {
      while (!feof (fe) ) {
        Personne* nouveau = new Personne();
        lireMot (fe, nouveau->nom);
        lireMot (fe, nouveau->prenom);
        booleen resu = insererDsTable (table, nouveau);
        if (!resu) {
          printf ("Débordement table\n");
        }
      }
      fclose (fe);
    }
    } break;

  case 2 : {
    printf ("Recherche séquentielle\n");
    Personne* cherche = lireNom (table);
    trouve = (Personne*) accesSequentiel (table, cherche);
    } break;

  case 3 : {
    printf ("Recherche séquentielle (sentinelle)\n");
    Personne* cherche = lireNom (table);
    trouve = (Personne*) accesSentinelle (table, cherche);
    } break;

  case 4 : {
    printf ("Recherche dichotomique récursive\n");
    Personne* cherche = lireNom (table);
    trouve = (Personne*) dichotomie (table, cherche);
    } break;

  case 5 : {
    printf ("Recherche dichotomique itérative\n");
    Personne* cherche = lireNom (table);
    trouve = (Personne*) dichotomieIter (table, cherche);
    } break;

  case 6 : {
    printf ("Numéro de l'élément recherché ? ");
    int i; scanf ("%d", &i); getchar();
    trouve = (Personne*) fournirElement (table, i);
    if (trouve==NULL) {
      printf ("Element numéro:%d inconnu\n", i);
    } else {
       ecrirePersonne (trouve);
    }
    } break;
```

```
    case 7 :
       listerTable (table);
       break;

    case 8 :
       trierTable (table) ;
       break ;
    }  // switch

    if ( (choix >= 2) && (choix <= 5) ) {
      if (trouve == NULL) {
        printf ("personne inconnue\n");
      } else {
        ecrirePersonne (trouve);
      }
    }
    if (!fini) {
      printf ("\nTaper Return pour continuer\n");
      getchar();
    }
  }  // while

  detruireTable (table);
}
#endif
```

4.1.7 Exemples d'application des tables

4.1.7.a Table de noms de personnes

Le fichier de données *personnes.dat* suivant est utilisé à titre d'exemple dans le choix 1 du menu précédent. Ce fichier doit être trié si on veut tester la recherche dichotomique. Pour le premier élément de la table, *Bouchard* est la clé ; celle-ci doit être unique (nom de login par exemple).

```
Bouchard  Jacques
Cabon     Amélie
Cosson    Sébastien
Dufour    Georges
Dupond    Marie
Duval     Matisse
Gautier   Renée
Jean      Robert
Legoff    Yann
Petit     Albert
Raboutin  Corentin
Robert    Michel
Tanguy    Monique
Xavier    Roland
Zazou     Chantal
```

Exemples de résultats :

```
LES TABLES

0 - Fin du programme
1 - Création à partir d'un fichier
```

```
    2 - Accès séquentiel à un élément
    3 - Accès séquentiel avec sentinelle
    4 - Accès dichotomique récursif
    5 - Accès dichotomique itératif
    6 - Accès au ième élément
    7 - Listage de la table
    8 - Tri de la table
    Votre choix ? 4
    Recherche dichotomique récursive
    Clé (nom) de la personne ? Gautier
    personne : Gautier Renée
```

4.1.7.b Table de noms de polynômes

Dans l'application sur les polynômes utilisant les listes (voir § 2.4.3.c, page 58), le programme principal de test ne gère qu'un seul polynôme pointé par po. Pour donner plus de généralité à ce programme, il faudrait créer une table contenant, pour chaque polynôme, le nom choisi par l'utilisateur, et la tête de la liste des monômes. Il faut déclarer un objet de type NomPoly, structure contenant le nom du polynôme et sa tête de liste.

Le programme de test de l'accès à la table peut s'écrire comme suit. Il faudrait développer un menu, ou mieux une interface graphique. Ce programme illustre bien la nécessité de découper une application en modules réutilisables. Le programme utilise le module des polynômes (*creerMonome()*, *creerPolynome()*, *insererEnOrdre()*, *listerPolynome()*, voir § 2.4.3.b, page 57), et le module des tables (type *Table*, *creerTable()*, *insererDsTable()*, *dichotomie()*).

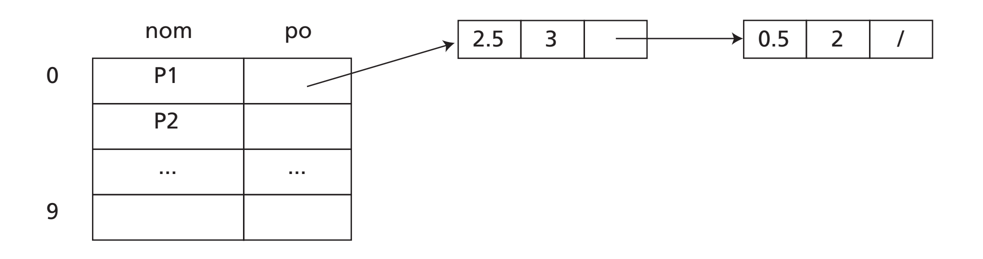


Figure 120 Table des noms de polynômes ($P1(x) = 2.5\ x^3 + 0.5\ x^2$) (sans les détails de l'implémentation).

La figure 120 indique les détails de l'implémentation de la table. L'utilisateur du module n'a pas besoin d'avoir conscience de tous ses détails. Il utilise seulement les fonctions de gestion des tables ou des listes.

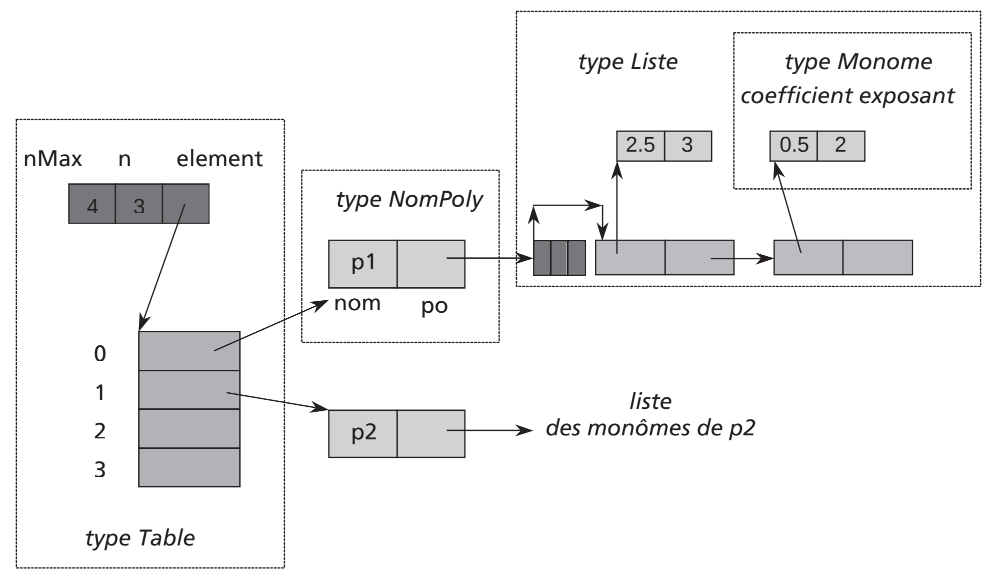


Figure 121 Détails de l'implémentation de la table.

```
// tablepolynome.cpp

#include <stdio.h>
#include <string.h>
#include "polynome.h"                        voir § 2.4.3.a, page 56
#include "table.h"                           voir § 4.1.6.a, page 206

// chaque élément de la table repère un objet de type NomPoly
typedef struct {
  char      nom[10];
  Polynome* po;
} NomPoly;  // nom du polynôme

NomPoly* creerNomPoly (char* nom) {
  NomPoly* nomPoly = new NomPoly();
  strcpy (nomPoly->nom, nom);
  nomPoly->po = creerPolynome();  // liste ordonnée
  return nomPoly;
}

// écriture et comparaison des noms de polynômes (table)
char* toStringPoly (Objet* objet) {
  NomPoly* nomPoly = (NomPoly*) objet;
  return nomPoly->nom;
}

int comparerPoly (Objet* objet1, Objet* objet2) {
  NomPoly* nomPoly1 = (NomPoly*) objet1;
  NomPoly* nomPoly2 = (NomPoly*) objet2;
  return strcmp (nomPoly1->nom, nomPoly2->nom);
}
```

```
// ajouter un nom de polynôme dans la table
void creerEntree (Table* table, char* nom) {
  NomPoly* nomPoly = creerNomPoly(nom);
  int resu = insererDsTable (table, nomPoly);
  if (!resu) printf ("Débordement de table\n");
}

void main () {
  #define NMAX 10
  Table*  table = creerTable (NMAX, toStringPoly, comparerPoly);

  // insérer des noms de polynômes dans la table des polynômes
  creerEntree (table, "p1");
  creerEntree (table, "p2");
  creerEntree (table, "p3");
  trierTable  (table);
  listerTable (table);

  // retrouver dans la table un polynôme à partir de son nom
  NomPoly* objetCherche = new NomPoly();
  strcpy (objetCherche->nom, "p2");
  NomPoly* trouve = (NomPoly*) dichotomie (table, objetCherche);
  if (trouve == NULL) {
    printf ("%s inconnu\n", "p2");
  } else {
    printf ("trouve : %s\n",  table->toString (trouve));

    // insérer des monômes au polynôme p2 en ordre décroissant
    Monome* nouveau = creerMonome(3, 3);
    insererEnOrdre (trouve->po, nouveau);
    nouveau = creerMonome(2, 2);
    insererEnOrdre (trouve->po, nouveau);
    nouveau = creerMonome(4, 4);
    insererEnOrdre (trouve->po, nouveau);

    printf ("Polynôme %s : ", "p2");
    listerPolynome (trouve->po);  // lister le polynôme de nom p2
    printf ("\n");
  }
}
```

Exemple de résultats :

```
 0 p1
 1 p2
 2 p3
trouve : p2
Polynôme p2 :  +4.00 x**4  +3.00 x**3  +2.00 x**2
```

4.2 VARIANTES DES TABLES

4.2.1 Rangement partitionné ou indexé

Si la table est sur disque, on peut fractionner la table en sous-tables de façon à limiter les accès disque. La première sous-table dite table majeure est amenée en mémoire centrale lors de l'ouverture du fichier. Les sous-tables sont ordonnées. Si

chaque sous-table contient 400 éléments (clé + pointeur de sous-table), on peut accéder à 160 000 éléments avec seulement 2 niveaux de table (voir Figure 122). La recherche à partir d'une clé demande une recherche dans la table majeure pour identifier quelle sous-table est concernée par l'élément cherché, une lecture de la sous-table de disque en mémoire centrale, une recherche dans la sous-table, et un accès direct au fichier de données. Le nombre de niveaux de sous-tables doit rester faible. Les recherches dans les sous-tables ordonnées peuvent être dichotomiques. On retrouve la partie tables d'index et la partie fichier de données. Cette structure conviendrait par exemple pour un dictionnaire du français de 100 000 mots (et leurs définitions) où il n'y a ni ajout ni retrait à faire.

Exemple :

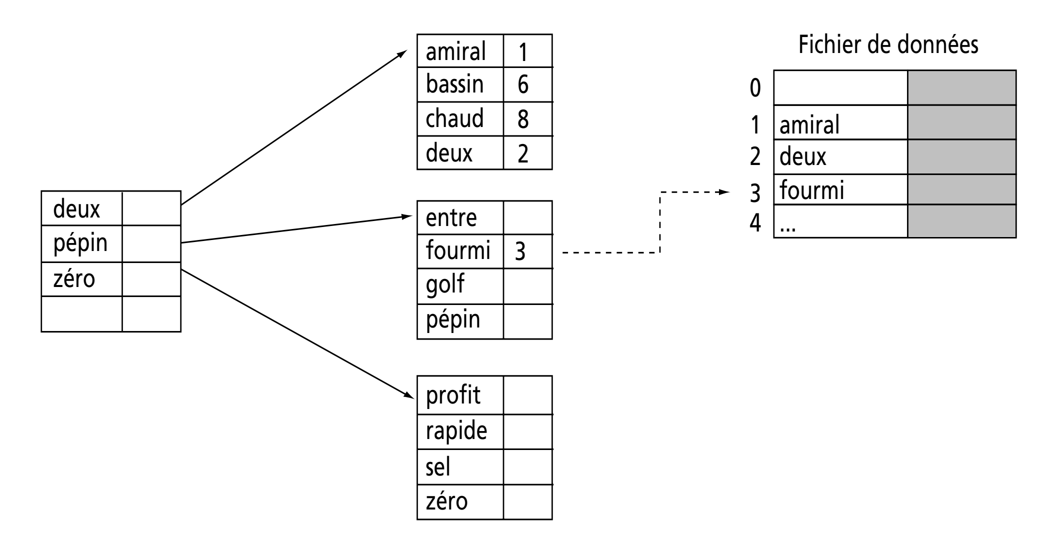


Figure 122 Arbre de tables d'index et fichier de données.

Séquentiel indexé

Si on doit faire des ajouts et des retraits, on peut ne remplir que partiellement les sous-tables de façon à laisser de la place pour les insertions. De toute façon, il faut prévoir une zone commune de débordement des sous-tables, avec chaînage des éléments débordant d'une même sous-table. Lorsqu'il y a trop d'éléments en zone de débordement, on peut réorganiser entièrement les tables d'index.

Une autre méthode, plus souvent utilisée maintenant, consiste à utiliser les techniques des B-Arbres (voir § 3.5, page 182). Les tables d'index constituent un arbre n-aire qui peut être géré comme un B-Arbre. Lorsqu'une sous-table est pleine, elle éclate en deux sous-tables. Si une sous-table ne contient plus assez d'éléments, elle fusionne avec une sous-table voisine. Il n'y a alors pas besoin de zone de débordement.

Avec cette technique, on peut accéder directement à une clé en utilisant les index mais on peut également parcourir les éléments séquentiellement.

4.2.2 Adressage calculé

L'évaluation d'une expression arithmétique peut dans certains cas donner directement le rang dans la table de l'élément cherché. Cela est possible lorsque la clé est structurée en sous-classes de tailles égales.

Exemple 1 : emploi du temps d'un établissement scolaire
On mémorise l'emploi du temps dans une table (en mémoire centrale ou sur disque) ce qui permet de faire des interrogations sur un enseignement particulier. Il y a cours du lundi (jour 1) au vendredi (jour 5). Il y a 8 groupes d'étudiants numérotés : A1, A2, B1, B2, C1, C2, D1, D2. Il y a 4 plages horaires numérotées : 1 de 8h-10 h, 2 de 10h-12h, 3 de 14h-16h, 4 de 16h-18h. Cet emploi du temps peut être schématisé comme indiqué sur la Figure 123 où les éléments sont rangés dans l'ordre jour, groupe, tranche horaire.

	0	1												31				159
Numéro du jour	1	1	1	1	1	1	1	1			1	1	1	1	2			
Groupe	A 1	A 1	A 1	A 1	A 2	A 2	A 2	A 2			D 2	D 2	D 2	D 2	A 1			
Tranche horaire	1	2	3	4	1	2	3	4			1	2	3	4	1			
Numéro enseignant																		
Numéro de matière																		
Numéro de salle																		

Figure 123 Table de l'emploi du temps.

La clé se constitue de la *concaténation* des numéros de jour, de groupe et de tranche horaire. Les numéros d'enseignant, de matière enseignée et de salle constituent la partie informations recherchées.

Exemple de recherche :
Comment retrouver les caractéristiques du cours du groupe *A2* du *mardi* de *14h à 16h* ? Si les indices commencent à 0, la clé se constitue de J=1 (2e jour), G=1 (2e groupe), T=2 (3e tranche horaire). Le rang est donné par l'équation suivante : *rang = J*32 + G*4 + T* soit 38. Il suffit d'accéder au 38e élément pour retrouver les caractéristiques du cours.

Si la table est en mémoire centrale, on accède à l'entrée 38 du tableau. Cependant, dans ce cas, on peut également considérer la table comme un tableau à 3 dimensions et accéder à un élément en donnant les indices J, G et T, laissant au

compilateur le soin de faire la conversion. Le programme ci-dessous permet de vérifier les 2 accès possibles. Les valeurs des numéros de jour, de groupe et de tranche horaire ne sont pas mémorisées dans la structure de type `cours` car elles sont implicites du fait de la séquentialité des valeurs (inutile de mémoriser les indices pour un tableau).

```
/* emploiDuTemps.cpp    tableau en mémoire centrale */

#include <stdio.h>

typedef struct {
  int nens;  // numéro d'enseignant
  int nmat;  // numéro de matière
  int nsal;  // numéro de salle
} Cours;

#define Mardi 1
#define GrpA2 1
#define TrCh3 2

#define MAXJOUR 5 // Nombre max de jours
#define MAXGROU 8 // Nombre max de groupes
#define MAXTRAN 4 // Nombre max de tranches horaires

void main () {
  Cours  epl [MAXJOUR][MAXGROU][MAXTRAN];
  char*  nomEns[] = {"Dupont", "Duval",   "Dufour"};
  char*  nomMat[] = {"Maths",  "Anglais", "Histoire"};
  char*  nomSal[] = {"Amphi",  "I115",    "M210"};

  // initialisation (partielle) du tableau
  Cours P = {1, 1, 2};
  epl [Mardi][GrpA2][TrCh3] = P;

  // recherche dans le tableau 1ère méthode
  printf ("\n%s ", nomEns [epl [Mardi][GrpA2][TrCh3].nens]);
  printf ("%s ",   nomMat [epl [Mardi][GrpA2][TrCh3].nmat]);
  printf ("%s ",   nomSal [epl [Mardi][GrpA2][TrCh3].nsal]);

  // recherche dans le tableau 2ième méthode
  Cours* debTab = (Cours*) epl;
  // 1*8*4 + 1*4 + 2 = 38
  printf ("\n%s ", nomEns [debTab[38].nens]);
  printf ("%s ",   nomMat [debTab[38].nmat]);
  printf ("%s ",   nomSal [debTab[38].nsal]);
}
```

Exemple 2 : fichier étudiants

Dans un établissement universitaire, il y a 3 départements (1:D1, 2:D2, 3:D3). Chaque département a deux promotions (1re et 2e année) d'au plus 160 étudiants. Un étudiant est caractérisé par son numéro de département, sa promotion et son numéro dans la promotion. Les étudiants pourraient être rangés par département, par promotion et numéro dans la promotion comme l'indique la Figure 124. Cependant, il y a **perte de place** (en grisé sur la figure) puisqu'il faut alors réserver 160 places par promotion, même si le nombre d'étudiants est inférieur. L'accès est rapide, au détri-

ment de l'espace mémoire occupé. Si les nombres d'étudiants sont très variables d'une promotion à l'autre, cette méthode n'est pas envisageable.

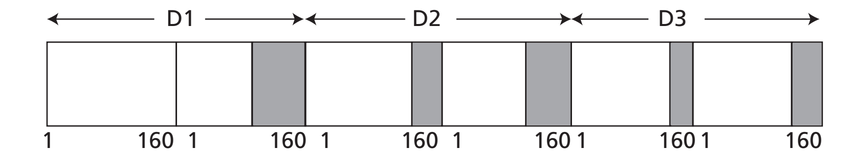


Figure 124 Table des étudiants (fichier en accès direct).

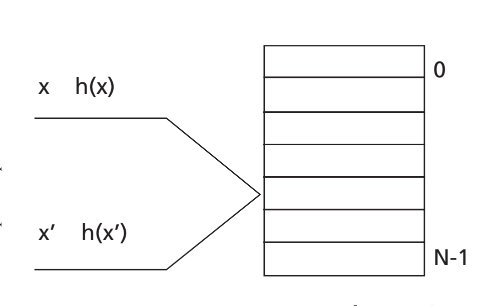
Le rang d'un étudiant connaissant son département D (de 1 à 3), sa promotion P (de 1 à 2) et son numéro dans la promotion N (de 1 à 160) est *rang = (D-1) * 320 + (P-1) * 160 + N-1*. Le premier étudiant a le rang 0.

4.3 ADRESSAGE DISPERSÉ, HACHAGE, HASH-CODING

4.3.1 Définition du hachage

Comme pour l'adressage calculé (voir § 4.2.2), il s'agit d'effectuer un calcul sur la clé qui doit indiquer la position de l'élément dans la table. Cependant, la fonction de calcul n'est plus injective ; deux clés différentes peuvent prétendre à la même place dans la table. Il faut donc arbitrer les conflits et définir une place pour tous les éléments. Le même calcul permet ensuite de retrouver un élément déjà rangé dans la table pour obtenir ses caractéristiques. Dans cette méthode, les éléments ne sont pas ordonnés dans la table.

Sur la Figure 125, à partir de la clé x, on définit une fonction h(x) qui engendre une valeur représentant le rang (l'indice) de cet indicatif dans la table ou dans le fichier. Il se peut que pour une autre clé x', la fonction h(x') fournisse la même place que pour h(x). x' est appelé synonyme de x ; on dit encore qu'il y a collision entre x et x'. Il faut trouver une autre place pour x'. Ainsi, si h("Dupond") vaut 25, "Dupont" est rangé à l'entrée 25 dans la table. Suivant la fonction h(x), il se peut que h("Duval") vaille également 25, d'où le conflit à résoudre.



– la fonction h n'est pas injective

– x est rangé en h(x)
x' est appelé synonyme de x
x' est en collision avec x

– il faut trouver une nouvelle place pour x' : c'est la résolution des collisions.

Figure 125 Principe du hachage.

h(x) fournit un rang dans la table compris entre 0 et N-1 pour **l'insertion** et pour la **recherche** de x. h(x) doit produire, à partir des différents x théoriquement possibles, des classes 0, ..., N-1 à peu près égales en nombre ; c'est-à-dire que les classes h(x) doivent être équiréparties même si les x ne le sont pas. S'il peut y avoir potentiellement C clés différentes, chaque entrée h(x) doit pouvoir en représenter C/N. Enfin, la fonction doit être rapide à calculer.

Afin de réduire les collisions, la table doit être plus grande que le nombre d'éléments à enregistrer (environ 2 fois plus). Là aussi, il y a perte de place mémoire en vue d'accélérer l'accès.

> ***Remarque*** : pour mieux illustrer les problèmes et leurs solutions sur les exemples suivants, les tailles des tables sont faibles (une vingtaine voire une cinquantaine d'entrées au plus). Dans la réalité, la méthode s'applique plutôt avec des ensembles de taille moyenne (milliers d'éléments) ou grande (centaines de milliers). Cependant, le principe reste le même.

4.3.2 Exemples de fonction de hachage

On effectue sur la clé des opérations arithmétiques et logiques qui produisent un nombre *place* compris entre 0 et N-1 (N : longueur de la table ou du fichier). S'il n'y a pas de collision (l'entrée *place* est libre), ce nombre permet de ranger la clé et ses caractéristiques dans l'entrée *place* de la table. Le même calcul permet de la retrouver en *place*. Quelques exemples de fonctions de hachage sont indiqués ci-dessous.

4.3.2.a Somme des rangs alphabétiques des lettres

h(x) = somme des rangs alphabétiques des lettres de x modulo N.

Exemple : h ("Dupond") ?

La somme des rangs alphabétiques ('a':1, ..., 'z':26) des lettres de *Dupond* est : 4+21+16+15+14+4 = 74. Si la taille de la table est N = 64, 74 modulo 64 vaut **10**, entrée attribuée à "Dupond" dans la table de 64 éléments.

Avec 10 caractères, la somme vaut au plus 260 pour “zzzzzzzzzz”. On ne peut donc gérer de grands volumes de données avec cette fonction qui est peu utilisée dans la pratique mais illustre bien le principe du hachage du point de vue pédagogique.

```
// somme des rangs alphabétiques des lettres de cle, modulo n
int hash1 (char* cle, int n) {
  int som = 0;
  for (int i=0; i<strlen (cle); i++) {
    if (isalpha (cle[i]) ) som += toupper (cle[i]) - 'A' +1;
  }
  return som % n;
}
```

Exemple d'appel : hash1 ("Dupond", 64); fournit 10.

4.3.2.b Addition des représentations binaires

On additionne la représentation binaire des caractères du mot, éventuellement en neutralisant minuscules et majuscules.

Exemple :

h (“Dupond”) = ‘D’ + ‘u’ + ‘p’ + ‘o’ + ‘n’ + ‘d’ vaut 618. On se ramène dans l’intervalle 0..N-1 par modulo N : 618 modulo 64 vaut 42 ; h (“Dupond”) = 42.

```
// somme des codes ascii des lettres de cle, modulo n
int hash2 (char* cle, int n) {
  int som = 0;
  for (int i=0; i<strlen(cle); i++) som += cle[i];
  return som % n;
}
```

Appel : hash2 ("Dupond", 64); fournit 42

4.3.2.c Méthode de la division

Cette méthode est la plus utilisée. Elle consiste à diviser la clé par la longueur N de la table et à considérer comme entrée, le reste de la division. N ne doit pas être quelconque : une division par 1 000 fournirait toujours les 3 derniers chiffres de la clé. Une division par 2^n consisterait à isoler les n derniers éléments binaires de la clé. Si N est un nombre premier, les différentes clés sont mieux réparties sur les diverses entrées de 0 à N-1 de la table.

Exemple :

Soit un fichier contenant 500 articles numérotés entre 000 000 000 et 999 999 999. La fonction de hachage h(x) = x modulo 997 (reste de la division par 997) fournit un h(x) compris entre 0 et 996 : 0 <= h(x) <= 996. Pour la clé 568419452, l’entrée est 568419452 divisée par 997 qui a pour reste 839.

```
// méthode de la division
int hash3 (long cle, int n) {
  return cle % n;
}
```

Appel : hash3 (568419452, 997); fournit 839.

4.3.2.d Changement de base

Soit un fichier de 500 articles et une table de 1 000 entrées. On considère que la clé est écrite en base 11 ; on la convertit en base 10 et on isole les trois derniers chiffres. Que vaut h (406327) ?

$$4 \times 11^5 + 6 \times 11^3 + 3 \times 11^2 + 2 \times 11 + 7 = 652582$$

On isole les 3 derniers chiffres 582 pour avoir un nombre entre 000 et 999. Le hash-code de h(406327) est 582.

```
// fonction récursive de conversion en base 10 du nombre n en base 11
long base11 (long n) {
  long q = n / 10;  // quotient
```

```
  long r = n % 10;  // reste
  if (q == 0) {
    return r;
  } else {
    return base11(q)*11+r;
  }
}

int hash4 (long cle, int n) {
  return base11 (cle) % n;
}
```

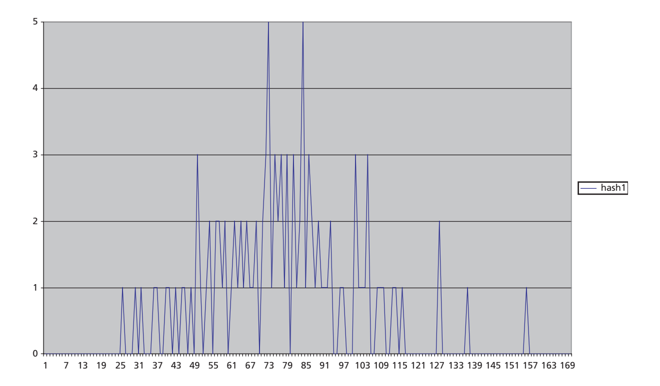
Appel : hash4 (406327, 1000); fournit 582.

4.3.3 Analyse de la répartition des valeurs générées par les fonctions de hachage

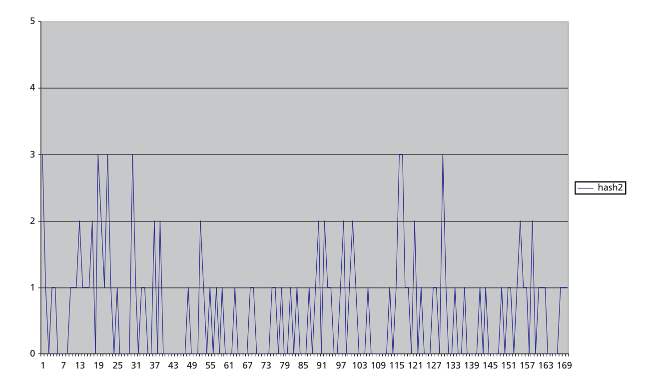
Remarques sur les fonctions de hachage : on peut envisager des opérations très diverses : extraire du nombre ou de la chaîne de caractères, un certain nombre d'éléments binaires que l'on concatène pour former un nouveau nombre ; élever la clé au carré et isoler un certain nombre d'éléments binaires, etc. Il faut seulement veiller à accéder à toutes les entrées de manière équirépartie. Si la fonction de hachage ne génère par exemple que des nombres pairs, une entrée sur deux de la table ne sera jamais sollicitée pour enregistrer une clé. Le nombre de collisions augmentera donc.

Les fonctions de hash-code hash1(), hash2() et une variante de hash1() appelée hash11() sont testées sur un fichier de 107 noms répartis dans une table de longueur 170 entrées.

La fonction **hash1**() (somme des rangs alphabétiques des caractères) n'est pas équirépartie. La fonction génère plus de valeurs de hash-code entre 50 et 100 sur l'exemple des 107 noms. Cinq noms ont pour hash-code 72 (Bouchard, Brunel, Etienne, Seznec, Tsemo). De même cinq noms ont pour hash-code 83. Par contre, aucun nom ne génére de hash-code entre 0 et 26, et seuls deux noms génèrent une valeur > 130.



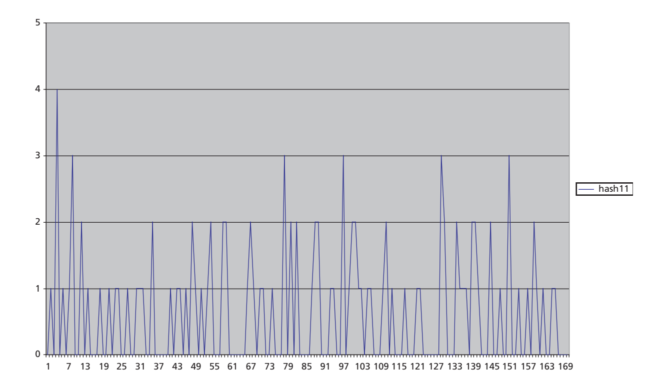
La fonction **hash2**() (somme des codes ascii des caractères) génère des valeurs mieux réparties sur l'ensemble des 170 entrées. Il y a au plus, sur l'exemple, 3 prétendants pour une entrée.



La fonction **hash11**() consiste à faire la somme modulo n des rangs des caractères alphabétiques et à multiplier ce rang par 97 fois l'indice du caractère. La dispersion est meilleure que pour hash1() sur l'ensemble des 170 entrées. Pour Dupond,

rang de d plus 1 fois 97 +

rang de u plus 2 fois 97 + etc.



4.3.4 Résolution des collisions

La clé x est rangée à l'adresse h(x) si l'entrée h(x) est libre. Si l'entrée est occupée, x est un synonyme, il faut chercher une autre entrée pour x : cette procédure s'appelle « résolution des collisions ». Il y a 2 variantes :

– la nouvelle entrée est donnée par une **nouvelle fonction** de résolution r(i), ou i désigne la ième tentative de résolution. L'entrée à tester pour la ième tentative est donc (N est la longueur de la table ; les entrées sont numérotées de 0 à N-1) : *(h(x) + r(i))   modulo N.*
– la nouvelle entrée est donnée par un **chaînage** qui fait référence à un élément :
  – d'une *table spéciale* regroupant les synonymes,
  – ou de la *même table*.

4.3.5 Résolution à l'aide d'une nouvelle fonction

La nouvelle fonction doit permettre de tester une et une seule fois, les différentes entrées de la table.

4.3.5.a Résolution r(i) = i

On cherche séquentiellement à partir de h(x), une entrée libre.

Exemple :

Soient les éléments suivants à insérer dans une table de N=26 entrées (de 0 à 25) et leur hash-code.

éléments :	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
hash-code :	0	0	2	0	3	9	9	25	25

Si l'entrée h(x) est occupée, il faut effectuer une ou plusieurs tentatives de résolution jusqu'à trouver une entrée libre. i indique la ième tentative de résolution. On effectue le calcul suivant : ((h(x) + i) modulo N) avec N = 26. Pour e1, la place 0 est disponible, e1 est rangé en 0. e2 devrait être en 0 mais la place est déjà prise. On effectue une première tentative de résolution (i=1) en calculant (h(x)+i) modulo 26, soit (0 + 1) modulo 26, soit 1. La clé e2 est rangée en 1, etc.

0	1	2	3	4	5	6	7	8	9	10	...	24	25
e1	e2	e3	e4	e5	e9				e6	e7			e8
0	0	2	0	3	25				9	9			25

e9 devrait être en 25, la place est prise par e8 rangé avant e9. On effectue les tentatives suivantes :

i = 1	(25 + 1)	mod 26	soit	0	occupé
i = 2	(25 + 2)	mod 26	soit	1	occupé
i = 3	(25 + 3)	mod 26	soit	2	occupé
i = 4	(25 + 4)	mod 26	soit	3	occupé
i = 5	(25 + 5)	mod 26	soit	4	occupé
i = 6	(25 + 6)	mod 26	soit	5	libre

La 6e tentative permet de trouver une entrée libre pour e9 qui est rangé dans l'entrée 5. Le nombre de collisions augmente au fur et à mesure que la table se remplit. Il peut se produire des points d'accumulation du fait que la résolution essaie de ranger sur les entrées qui suivent l'entrée indiquée par la fonction de hachage comme le montre le tableau précédent : il y a 3 prétendants pour la place 0 en début de table. Les valeurs de hash-code sont indiquées sur le schéma pour vérification ; elles ne sont pas mémorisées dans la table. La fonction resolution1() calcule l'entrée à essayer pour un hash-code H dans une table de longueur N pour la ième tentative.

```
// résolution r(i) = i
int resolution1 (int h, int n, int i) {
  return (h+i) % n;
}
```

La séquence suivante fournit les entrées à essayer (jusqu'à trouver une entrée libre) pour un hash-code H=2 dans une table de longueur N=26 (entrées de 0 à 25). Les entrées suivant H=2 sont essayées successivement en considérant la table comme circulaire ; le suivant de l'entrée 25 est l'entrée 0. Toutes les entrées sont testées une et une seule fois jusqu'à trouver une entrée libre. Si aucune entrée libre n'est trouvée, c'est que la table est pleine.

```
// résolution séquentielle r(i) = i
for (i=1; i<=25; i++) {
  printf ("%3d", resolution1 (2, 26, i) );
}
```

```
Résultats :
3 4 5 6 7 8 9 10 11 13 14 15 16 17 18 19 20 21 22 23 24
25 0 1 2
```

Nombre d'accès à la table pour retrouver un élément :
1 : e1, e3, e6, e8;   2 : e2, e5, e7;   4: e4;   7 : e9.

*4.3.5.b Résolution r (i) = K * i*

On essaie de disperser les points d'accumulation en éloignant les synonymes de K entrées les uns des autres. La fonction de résolution est donc : r (i) = K * i. Lors de la ième tentative, l'entrée à essayer est donnée par (h(x) + K * i) modulo N. K et N doivent être premiers entre eux : ils ne doivent avoir que 1 comme diviseur commun.

Exemple :

Ranger les identificateurs de l'exemple précédent avec K = 5 et N = 26. 5 et 26 sont premiers entre eux (pas de diviseur commun).

éléments :	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
hash-code :	0	0	2	0	3	9	9	25	25

0	1	2	3	4	5	6	7	8	9	10	...	14	15	...	25
e1		e3	e5	e9	e2				e6	e4		e7			e8
0		2	3	25	0				9	0		9			25

pour $\mathbf{e_9}$

	h(x) + K * i	modulo N			
i = 1	25 + 5 * 1	modulo 26	soit	4	libre

La clé e9 est rangée dans l'entrée 4.

```
// résolution r(i) = k*i
int resolution2 (int h, int n, int i) {
  int k = 5; // ou par exemple, le premier avec n qui précède n
  return (h+k*i) % n;
}
```

Sur l'exemple, on diminue les points d'accumulation en début de table. Si la table est de longueur N, la résolution doit permettre de tester toutes les entrées une fois et

une seule fois (donc générer des nombres de 0 à N-1). La séquence suivante fournit les entrées à essayer (jusqu'à trouver une entrée libre) pour un hash-code H=2 dans une table de longueur N=26 entrées (de 0 à 25), en progressant de K=5 entrées à chaque tentative.

```
// résolution par pas de K, K=5 et N=26 premiers entre eux
for (i=1; i<=25; i++) {
  printf ("%3d", resolution2 (2, 26, i) );
}
```

Pour un hash-code H=2, les entrées sont essayées une et une seule fois dans l'ordre : 7, 12, 17, 22, 1, 6, 11, 16, 21, 0, 5, 10, 15, 20, 25, 4, 9, 14, 19, 24, 3, 8, 13, 18, 23. On progresse de 5 en 5 circulairement.

Nombre d'accès à la table pour retrouver un élément :
1 : e1, e3, e5, e6, e8;   2 : e2, e7, e9;   3: e4.

K et N doivent être premiers entre eux, sinon, toutes les entrées ne sont pas essayées. La séquence suivante fournit les entrées à essayer (jusqu'à trouver une entrée libre) pour un hash-code H=2 dans une table de longueur N=10 entrées (numérotées de 0 à 9), en progressant de K=5 entrées à chaque tentative. K et N ont un diviseur commun 5.

```
// résolution avec K=5 et N=10 non premiers entre eux
for (i=1; i<=9; i++) {
  printf ("%3d", resolution2 (2, 10, i) );
}
```

Pour un hash-code H=2, seules les entrées 7 et 2 sont essayées. La séquence générée est : 7, 2, 7, 2, 7, 2, 7, 2, 7.

*4.3.5.c Résolution quadratique r(i) = i * i (N : nombre premier)*

On progresse suivant le carré de i, ième tentative pour trouver une place pour un élément en collision. La fonction *resolution3()* fournit l'entrée à essayer pour un élément de hash-code H dans une table de longueur N, lors de la ième tentative. N doit être un nombre premier.

Exemple

Ranger les identificateurs de l'exemple précédent (table de longueur N = 29).
éléments et leur hash-code.

e1	e2	e3	e4	e5	e6	e7	e8	e9	e10
0	0	2	0	3	9	9	25	25	25

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**...**	**15**	**...**	**25**	**26**	**27**	**28**
e1	e2	e3	e5	e4	e10				e6	e7					e8	e9		
0	0	2	3	0	25				9	9					25	25		

pour $\mathbf{e_{10}}$

	h(x) + i * i	modulo N			
i = 1	25 + 1 * 1	modulo 29	soit	26	occupé
i = 2	25 + 2 * 2	modulo 29	soit	0	occupé
i = 3	25 + 3 * 3	modulo 29	soit	5	libre

La clé e10 est rangée dans l'entrée 5.

```
// résolution r(i) = i*i
int resolution3 (int h, int n, int i) {
  return (h+i*i) % n;
}
```

La boucle suivante permet de calculer les différentes entrées essayées pour un élément de hash-code H=2 dans une table de longueur N=29.

```
// résolution quadratique en i*i
for (i=1; i<=28; i++) {
  printf ("%3d", resolution3 (2, 29, i) );
}
```

Résultats de la boucle précédente :

```
3, 6, 11, 18, 27, 9, 22, 8, 25, 15, 7, 1, 26, 24, 26, 7,
1, 15, 25, 8, 22, 9, 27, 18, 11, 6, 3
```

Les résultats montrent que seulement la moitié de la table peut être accédée car la séquence générée est symétrique par rapport à l'élément du milieu 24. Ceci peut être démontré mathématiquement. Si N est grand, l'accès à seulement la moitié des éléments de la table pour loger un nouvel élément n'est pas réellement pénalisant.

Nombre d'accès à la table pour retrouver un élément :
1 : e1, e3, e5, e6, e8;   2 : e2, e7, e9;   3: e4;   4 : e10.

4.3.5.d Résolution pseudo-aléatoire (N : puissance de 2)

L'idée reste la même de disperser les synonymes sur toute la table, mais en évitant les régularités comme précédemment où les synonymes sont répartis de K en K entrées ou d'un pas variable dépendant du carré de i. On fait appel à un générateur de nombres pseudo-aléatoires r(i) = aleat(i) compris entre 1 et N-1, générés une et une seule fois, de façon à répartir les synonymes sur toute la table et éviter les points d'accumulation. La séquence est pseudo-aléatoire, car c'est toujours la même séquence (pour une longueur de table N donnée) pour ranger l'identificateur et pour le retrouver. La longueur de la table doit être une puissance de 2.

Exemple:

La séquence r(i) suivante est générée par le générateur de nombres pseudo-aléatoires pour N=16 : 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3

ranger les éléments :	e1	e2	e3	e4	e5
hash-code :	1	0	2	1	1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e2	e1	e3					e4						e5		
0	1	2					1						1		

pour e4	h (x) + r (i)	modulo 16			
i = 1	1 + 1	modulo 16	soit	2	occupé
i = 2	1 + 6	modulo 16	soit	7	libre pour e4

pour e5	h (x) + r (i)	modulo 16			
i = 1	1 + 1	modulo 16	soit	2	occupé
i = 2	1 + 6	modulo 16	soit	7	occupé
i = 3	1 + 15	modulo 16	soit	0	occupé
i = 4	1 + 12	modulo 16	soit	13	libre pour e5

Nombre d'accès à la table pour retrouver un élément :

1 : e1, e2, e3; 3 : e4; 5 : e5.

Le programme suivant permet de générer une et une seule fois des nombres pseudo-aléatoires compris entre 0 et N-1. reInit est vrai s'il faut réinitialiser le générateur de nombres pseudo-aléatoires, c'est-à-dire recommencer la séquence.

```
// fournit une et une seule fois des nombres pseudo-aléatoires
// entre 0 et n-1 inclus
// programme donné sans démonstration (pour test)
// n doit être une puissance de 2 (4, 8, 16, ..., 256, etc.)
int aleat (int n, booleen reInit) {
  static  int r = 1;
  static int n4 = n*4;
  if (reInit) {
    n4 = n*4;
    r = 1;
    return r;
  } else {
    r *= 5;
    r %= n4;
    // printf ("\naleat  %2d\n", r/4);
    return r / 4;
  }
}

// réinitialiser le générateur de nombres pseudo-aléatoires
void initaleat (int n) {
  aleat (n, vrai);
}

// résolution pseudo-aléatoire
int resolution4 (int h, int n, int i) {
  return ( h + aleat (n, faux) ) % n;
}
```

La séquence suivante fournit les entrées à essayer (jusqu'à trouver une entrée libre) pour un hash-code 1 dans une table de 16 entrées (numérotées de 0 à 15), en progressant de **aleat**() entrées à chaque tentative.

```
initaleat (16);  // réinitialiser le générateur de nombres aléatoires
for (i=1; i<=15; i++) {
  printf ("%3d", resolution4 (1, 16, i) );
}
```

Pour un hash-code 1, les entrées sont essayées une et une seule fois dans l'ordre 2, 7, 0, 13, 14, 3, 12, 9, 10, 15, 8, 5, 6, 11, 4.

4.3.6 Le fichier d'en-tête des fonctions de hachage et de résolution

Les fonctions de hachage et de résolution peuvent être regroupées dans les fichiers *fnhc.h* et *fnhc.cpp*.

```
/* fnhc.h  fonctions de hachage et de résolution */

#ifndef FNHC_H
#define FNHC_H

typedef int     booleen;
#define faux    0
#define vrai    1

typedef void Objet;

int hash1 (Objet* objet, int nMax);     // somme des rangs alphabétiques
int hash2 (Objet* objet, int nMax);     // somme des codes ascii
int hash3 (Objet* objet, int nMax);     // division par nMax
int hash4 (Objet* objet, int nMax);     // base 11

int resolution1 (int h, int n, int i);  // i
int resolution2 (int h, int n, int i);  // k*i
int resolution3 (int h, int n, int i);  // i*i
int resolution4 (int h, int n, int i);  // pseudo-aléatoire

#endif
```

4.3.7 Le corps du module sur les fonctions de hahscode et de résolution

D'une manière générale, la fonction de hash-code est passée en paramètre lors de la déclaration de la table de hash-code. Elle opère sur un objet. La fonction de hash-code peut aussi, si besoin est, être définie dans le programme d'application.

```
/* fnhc.cpp fonctions de hash-code et résolution */

#include <stdio.h>
#include <stdlib.h>  // abs
#include <string.h>  // strlen
#include <ctype.h>   // isalpha
```

```
#include "fnhc.h"

int hash1 (char* cle, int n)                              voir 4.3.2.a
int hash2 (char* cle, int n)                              voir 4.3.2.b
int hash3 (long  cle, int n)                              voir 4.3.2.c
long base11 (long n)                                      voir 4.3.2.d
int hash4 (long  cle, int n)                              voir 4.3.2.d

int hash1 (Objet* objet, int n) {
  return hash1 ((char*) objet, n);
}

int hash2 (Objet* objet, int n) {
  return hash2 ((char*) objet, n);
}

int hash3 (Objet* objet, int n) {
  long* pcle =  (long*) objet;
  return hash3 ( *pcle, n);
}

int hash4 (Objet* objet, int n) {
  long* pcle =  (long*) objet;
  return hash4 (*pcle, n);
}

int resolution1 (int h, int n, int i)                     voir ci-dessus
int resolution2 (int h, int n, int i)
int resolution3 (int h, int n, int i)
int resolution4 (int h, int n, int i)
```

4.3.8 Le type TableHC (table de hachage)

Le type *TableHC* est décrit dans le fichier *tablehc.h* suivant. Il comprend, comme pour le type *Table* définit au § 4.1.6.a, page 206, le nombre maximum d'éléments dans la table, le nombre réel d'éléments, et un tableau de pointeurs sur les objets du tableau. 4 fonctions sont passées en paramètre (écriture et comparaisons des objets de la table, hash-code et résolution).

Si on veut pouvoir effectuer des destructions, il faut distinguer 3 états : libre, occupé, ou détruit. Un élément est déclaré ne pas appartenir à la table si l'entrée proposée par la résolution est libre. En cas de retrait d'un élément, il ne faut pas rompre cette liste implicite des éléments occupés. L'entrée est marquée détruite (et non libre), ce qui permet une insertion, mais n'arrête pas la recherche d'un élément. Les cas de retraits ne sont pas envisagés dans la suite des algorithmes, mais laissés en exercice.

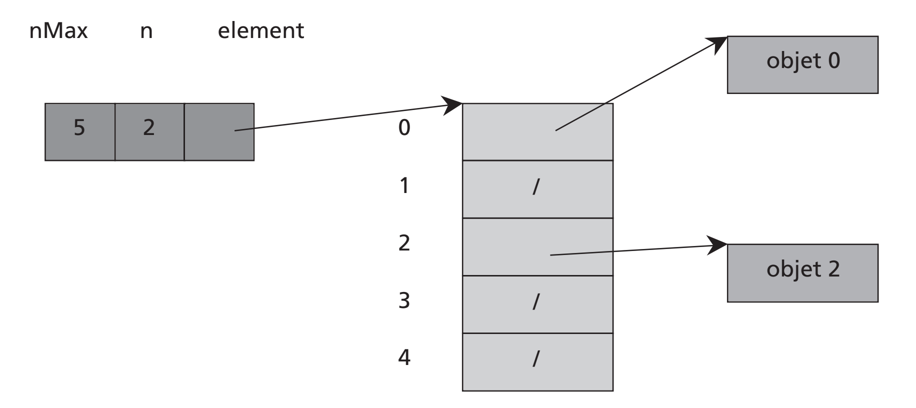


Figure 126 Le type TableHC (table de hachage). Les objets ont une place attribuée par une fonction de hash-code et une fonction de résolution des collisions. Les pointeurs des objets ne sont pas consécutifs en mémoire.

```
/* tableHC.h sans chainage des synonymes */

#ifndef TABLEHC_H
#define TABLEHC_H

#include "fnhc.h"

typedef void Objet;

typedef struct {
  int     nMax;          // nombre max (longueur) de la table
  int     n;             // nombre d'éléments dans la table
  Objet** element;       // tableau de pointeurs sur les objets
  char*   (*toString)   (Objet*);
  int     (*comparer)   (Objet*, Objet*);
  int     (*hashcode)   (Objet*, int);
  int     (*resolution) (int, int, int);
} TableHC;

TableHC* creerTableHC      (int nMax, char* (*toString) (Objet*),
                            int (*comparer) (Objet*, Objet*),
                            int (*hashcode) (Objet*, int),
                            int (*resolution) (int, int, int));
TableHC* creerTableHC      (int nMax);
booleen  insererDsTable    (TableHC* table, Objet* nouveau);
Objet*   rechercherTable   (TableHC* table, Objet* objetCherche);
void     listerTable       (TableHC* table);
int      nbAcces           (TableHC* table, Objet* objetCherche);
double   nbMoyAcces        (TableHC* table);
void     listerEntree      (TableHC* table, int entree);
void     ordreResolution   (TableHC* table, int entree);
#endif
```

4.3.8.a Création d'une table de hachage

L'espace de la table et des éléments du tableau est alloué dynamiquement comme précédemment (voir § 4.1.3, page 200). La fonction *creerTableHC()* mémorise en plus les fonctions de hachage et de résolution.

```
// création d'une table de hashcode de nMax entrées
TableHC* creerTableHC (int nMax, char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*),
                                           int (*hashcode) (Objet*, int),
                                     int (*resolution) (int, int, int)) {
  TableHC* table    = new TableHC();
  table->nMax       = nMax;
  table->n          = 0;
  table->element    = (Objet**) malloc (sizeof(Objet*) * nMax);
  table->toString   = toString;
  table->comparer   = comparer;
  table->hashcode   = hashcode;
  table->resolution = resolution;
  return table;
}

TableHC* creerTableHC (int nMax) {
  return creerTableHC (nMax, toChar, comparerCar, hash1, resolution1);
}
```

La fonction : *static int resolution (TableHC* table, int h) ;* cherche une entrée libre dans la table pour un élément de hash-code h. Cette fonction fournit la place (entier) attribuée à l'élément en collision ou –1 si la table est saturée. Si la résolution est pseudo-aléatoire, il faut réinitialiser le générateur de nombres, de façon à recommencer au début de la séquence de nombres. L'algorithme est simple : tant qu'on n'a pas effectué nMax-1 tentatives et tant qu'on n'a pas trouvé une entrée libre (ou détruite), on appelle la fonction de résolution (passée en paramètre lors de l'appel de *creerTableHC()*) pour connaître la prochaine entrée à tester. Cet algorithme est le même quelles que soient les fonctions de hachage et de résolution.

```
// fournir l'entrée réellement attribuée pour un hashcode h
// en appliquant la résolution de la table;
// fournit -1 en cas d'échec
static int resolution (TableHC* table, int h) {
  booleen trouve = faux;
  int     i = 1;     // ième tentative
  int     re;
  initaleat (table->nMax);  // si la résolution est aléatoire

  while ((i<table->nMax) && !trouve) {
    re = table->resolution (h, table->nMax, i);
    trouve = table->element[re] == NULL;
    i++;
  }
  if (!trouve) re = -1;
  return re;
}
```

4.3.8.b Ajout d'un élément dans une table de hachage

La fonction *insererDsTable()* insère l'élément pointé par *nouveau* dans la table (voir § 4.1.6.a, page 206). Si l'entrée h désignée par la fonction de hachage est libre, l'élément est rangé en h, sinon, on fait appel à la fonction de résolution qui attribue une entrée *re* pour l'élément nouveau à insérer, ou –1 si la fonction de résolution n'a pas trouvé de place libre. La fonction retourne vrai s'il y a eu insertion, et faux sinon. Le nombre d'éléments dans la table est incrémenté de 1 en cas de succès.

```
// insérer l'objet nouveau dans la table;
// fournir faux si la table est saturée
booleen insererDsTable (TableHC* table, Objet* nouveau) {
  int h = table->hashcode (nouveau, table->nMax);
  if (table->element[h] == NULL) {
    table->element[h] = nouveau;
  } else {
    int re = resolution (table, h);
    if (re != -1) {
      table->element[re] = nouveau;
    } else {
      printf ("insererDsTable saturée hashcode %3d pour %s\n",
               h, table->toString(nouveau));
      return faux;
    }
  }
  table->n++;
  return vrai;
}
```

4.3.8.c Recherche d'un élément dans une table de hachage

La fonction *rechercherTable()* fournit un pointeur sur l'objet cherché (*objetCherche*) de la table. Si l'élément n'est pas à l'entrée indiquée par son hash-code *hc*, on le cherche en examinant successivement les entrées *re* données par la fonction de résolution. Si l'entrée *re* est libre et que l'élément n'a toujours pas été trouvé, c'est que l'élément n'est pas dans la table. Cette recherche s'apparente aux différentes recherches vues précédemment pour le type Table. La fonction retourne un pointeur sur l'objet cherché, ou NULL si l'objet n'existe pas dans la table.

```
// rechercher objetCherche dans la table
Objet* rechercherTable (TableHC* table, Objet* objetCherche) {
  booleen trouve = faux;
  int hc = table->hashcode (objetCherche, table->nMax);
  int re = hc;
  int i  = 1;
  while ( (i<table->nMax) && !trouve && (re != -1) ) {
    if (table->element[re] == NULL) {
        re = -1;
    } else {
      trouve = table->comparer (objetCherche, table->element[re]) == 0;
      if (!trouve) re = table->resolution (hc, table->nMax, i++);
    }
  }
  return re==-1 ? NULL : table->element[re];
}
```

4.3.8.d Listage de la table

La fonction *listerTable()* liste les entrées occupées de la table de hash-code. Elle indique également le nombre moyen d'accès pour retrouver un élément dans la table.

```
// lister la table
// et calculer le nombre moyen d'accès pour retrouver un élément
void listerTable (TableHC* table) {
  int sn = 0;
  for (int i=0; i<table->nMax; i++) {
    if (table->element[i] != NULL) {
      printf ("%3d : hc:%3d %s\n", i,
         table->hashcode (table->element[i], table->nMax),
         table->toString (table->element[i]));
      int n = nbAcces (table, table->element[i]);
      if (n>0) sn += n;
    }
  }
  printf ("\nNombre d'éléments dans la table : %d", table->n);
  printf ("\nTaux d'occupation de   la table : %.2f",
          table->n / (double) table->nMax);
  printf ("\nNombre moyen d'accès à la table : %.2f\n\n",
           sn / (double) table->n);
}
```

4.3.8.e Nombre moyen d'accès

nbAcces() fournit le nombre d'accès pour retrouver un élément de la table, ou -1 si l'élément n'est pas dans la table.

```
// fournir le nombre d'accès à la table
// pour retrouver objetCherche; -1 si inconnu
int nbAcces (TableHC* table, Objet* objetCherche) {
  int na = 0;      // nombre d'accès
  int hc = table->hashcode (objetCherche, table->nMax);
  if (table->element[hc] == NULL) {
    na = -1;       // élément inconnu
  } else {
    int re = hc;   // résolution
    int i  = 1;    // ième tentative
    na++;
    initaleat (table->nMax);  // si la résolution est aléatoire
    while ( table->comparer (objetCherche, table->element[re]) != 0 ) {
      na++;
      re = table->resolution (hc, table->nMax, i++);
      if (table->element[re] == NULL) return -1; // élément inconnu
    }
  }
  return na;
}
```

nbMoyAcces() fournit le nombre moyen d'accès pour retrouver un élément de la table.

```
// nombre moyen d'accès
double nbMoyAcces (TableHC* table) {
  int sn = 0;
  for (int i=0; i<table->nMax; i++) {
    if (table->element[i] != NULL) {
```

```
      int n = nbAcces (table, table->element[i]);
      if (n>0) sn += n;
    }
  }
  return sn / (double) table->n;
}
```

4.3.8.f Fonction de contrôle des emplacements

La fonction *listerEntree()* liste, à titre indicatif ou de mise au point, pour une entrée donnée, les éléments à parcourir lorsqu'un élément n'est pas dans la table. La fonction fournit les clés et les hash-codes des éléments rencontrés.

```
// lister les éléments à parcourir pour insérer
// un nouvel élément de hash-code entree
void listerEntree (TableHC* table, int entree) {
  printf ("\nentrée à parcourir pour hashcode %d\n", entree);
  if (table->element[entree] == NULL) {
    printf ("aucun objet de hash-code %d\n", entree);
  } else {
    int i  = 1;
    int re = entree;
    while (table->element[re] != NULL) {
      printf ("%3d %3d : hc:%3d %s\n", i, re,
         table->hashcode (table->element[re], table->nMax),
         table->toString (table->element[re]));
      re = table->resolution (entree, table->nMax, i++);
    }
  }
}
```

4.3.8.g Ordre de la résolution

La fonction *ordreResolution()* fournit l'ordre de recherche d'une entrée libre en partant d'une entrée donnée. Si la résolution est aléatoire, il faut réinitialiser le générateur de nombre aléatoire.

```
// l'ordre des nMax entrées essayées en cas de conflit
// pour un hashcode "entree"
void ordreResolution (TableHC* table, int entree) {
  printf ("\nordre des résolutions pour l'entrée %d\n", entree);
  initaleat (table->nMax);                      voir § 4.3.5.d, page 225
  for (int i=1; i<table->nMax; i++) {
    printf ("%2d ", table->resolution (entree, table->nMax, i));
  }
  printf ("\n");
}
```

4.3.9 Exemple simple de mise en œuvre du module sur les tables de hash-code

Le programme suivant déclare une table de hash-code utilisant la fonction *hash1()* de calcul du hash-code et la fonction de résolution *resolution1()*. Ces fonctions pourraient être changées et définies dans le programme appelant si les fonctions prédéfinies ne conviennent pas.

```
void main () {
  Personne* p1 = creerPersonne ("Dupond", "Jacques");
  Personne* p2 = creerPersonne ("Dufour", "Albert");
  Personne* p3 = creerPersonne ("Duval",  "Marie");
  Personne* p4 = creerPersonne ("Ponddu", "Jacques");
  Personne* p5 = creerPersonne ("Punddo", "Jacques");

  // table de Personne                             voir en 2.4.1, page 51
  TableHC* table = creerTableHC (16, toStringPersonne, comparerPersonne,
                                                  hash1, resolution1);

  insererDsTable (table, p1);
  insererDsTable (table, p2);
  insererDsTable (table, p3);
  insererDsTable (table, p4);
  insererDsTable (table, p5);

  listerTable (table);

  printf ("\nrecherche de la personne Dupond\n");
  Personne* cherche = creerPersonne ("Dupond", "?");
  Personne* trouve = (Personne*) rechercherTable (table, cherche);
  printf ("trouve %s\n", toStringPersonne (trouve));

  printf ("\nrecherche de la personne Punddo\n");
  cherche = creerPersonne ("Punddo", "?");
  trouve  = (Personne*) rechercherTable (table, cherche);
  printf ("trouve %s\n", toStringPersonne (trouve));
}
```

La table de hash-code de 16 éléments (hash1 et résolution1) : Dupond, Ponddu et Punddo sont synonymes (hc : 10).

```
 0 :
 1 :
 2 :
 3 :
 4 :
 5 : hc:  5   1 Dufour Albert
 6 :
 7 :
 8 :
 9 :
10 : hc: 10   1 Dupond Jacques
11 : hc: 10   2 Ponddu Jacques
12 : hc: 12   1 Duval Marie
13 : hc: 10   4 Punddo Jacques
14 :
15 :
```

La recherche de Dupond trouvé directement à l'entrée 10

```
recherche de la personne Dupond
rechercherTable re : 10 occupé par Dupond Jacques
trouve Dupond Jacques
```

La recherche de Punddo trouvé après avoir consulté 10, 11, 12 et 13 (résolution r(i)=i).

```
recherche de la personne Punddo
rechercherTable re : 10 occupé par Dupond Jacques
rechercherTable re : 11 occupé par Ponddu Jacques
rechercherTable re : 12 occupé par Duval Marie
rechercherTable re : 13 occupé par Punddo Jacques
trouve Punddo Jacques
```

4.3.10 Programme de test des fonctions de hachage

Le menu suivant permet de tester les fonctions de hachage définies ci-dessus, ainsi que les diverses résolutions, et fonctions de gestion de la table.

```
TABLE (HASH-CODING)

 0 - Fin
 1 - Initialisation de la table
 2 - Hash-code d 'un élément
 3 - Ordre de test des N-1 entrées
 4 - Ajout d'un élément dans la table
 5 - Ajout d'éléments à partir d'un fichier
 6 - Liste de la table
 7 - Recherche d'une clé
 8 - Collisions à partir d'une entrée

Votre choix ? 1
```

Le choix 1 permet de préciser les paramètres de la table (longueur, fonctions de hachage et de résolution). Le choix 2 permet de calculer le hash-code d'un élément à fournir en fonction des paramètres de la table. Le choix 3 indique pour une entrée H donnée, l'ordre dans lequel seront examinées les N-1 entrées restantes lors de la résolution. Le choix 4 ajoute un élément dans la table, alors que le choix 5 ajoute des éléments lus dans un fichier. Le choix 6 liste les entrées occupées de la table. Le choix 7 permet de retrouver un élément dans la table. Le choix 8 indique les différentes tentatives pour trouver une entrée libre, en précisant pour chaque entrée consultée, l'élément et son hash-code.

```
Votre choix ? 1

Paramètres
Longueur N de la table ? 26
Fonctions de hachage
  1 somme des rangs alphabétiques
```

```
    2 division par N
    3 somme des caractères ascii
    4 changement de base
  Votre choix ? 1
  Résolution
    1 r(i) = i
    2 r(i) = K*i
    3 r(i) = i*i
    4 pseudo-aléatoire
  Votre choix ? 1
```

Exemple du choix 6, correspondant à l'exemple de la fonction de hachage hash1(), et à la résolution linéaire pour une table de longueur 26 (voir § 4.3.5.a, page 222). La table a été créée à partir du fichier *cles.dat* suivant correspondant aux exemples de résolution pour les fonctions r(i) = i, k*i et i*i vues précédemment. Z (nommé e1 sur les exemples) a pour hash-code 0, ainsi que ZZ(e2) et ZZZ(e3), etc.

```
Z    e1(0)                                fichier cles.dat
ZZ   e2(0)
B    e3(2)
ZZZ  e4(0)
C    e5(3)
I    e6(9)
IZ   e7(9)
Y    e8(25)
YZ   e9(25)
```

Listage de la table après création :

```
  Votre choix ? 6                               cas r(i)=i
    0 : hc:  0   1 Z e1(0)
    1 : hc:  0   2 ZZ e2(0)
    2 : hc:  2   1 B e3(2)
    3 : hc:  0   4 ZZZ e4(0)
    4 : hc:  3   2 C e5(3)
    5 : hc: 25   7 YZ e9(25)
    6 :
    7 :
    8 :
    9 : hc:  9   1 I e6(9)
   10 : hc:  9   2 IZ e7(9)
   11 :
   12 :
   13 :
   14 :
   15 :
   16 :
   17 :
```

```
        18 :
        19 :
        20 :
        21 :
        22 :
        23 :
        24 :
        25 : hc: 25   1 Y e8(25)
```

Remarque : dans la réalité, les tables de hachage contiennent des milliers de valeurs. L'exemple est seulement pédagogique.

Exercice 24 - Menu pour une table de hachage

Écrire le programme principal correspondant au menu donné précédemment pour la mise en œuvre des fonctions de hachage.

4.3.11 Résolution par chaînage avec zone de débordement

Cette fois, les éléments en collision (les synonymes) sont chaînés entre eux. Pour la recherche, il suffit de parcourir la liste pour retrouver un élément.

4.3.11.a Avec une table séparée pour les synonymes

Les synonymes sont chaînés dans une zone à part de débordement allouée dynamiquement ou statiquement.

Allocation statique de la table et allocation dynamique de la zone de débordement. On crée une liste des éléments ayant même hash-code (voir Figure 127).

Exemple

Ranger les identificateurs suivants (longueur de la table n = 26) :

éléments :	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9
hash-code :	0	0	2	0	3	9	9	25	25

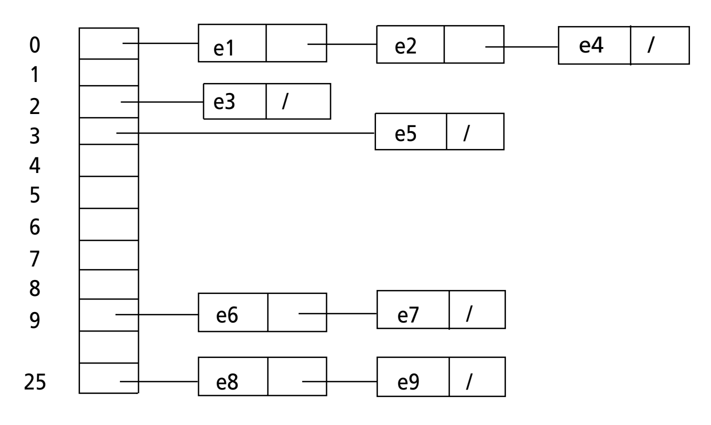


Figure 127 Chaînage en allocation dynamique des synonymes.

Allocation statique (en mémoire centrale ou sur disque). La table ou le fichier en accès direct est alloué en début d'exécution (voir Figure 128). La zone des synonymes suit la table principale. La table principale a des entrées de 0 à N-1 directement accédées à partir du hash-code. Si l'élément n'est pas dans l'entrée fournie par le hash-code, il faut parcourir la liste des synonymes commençant dans le 3[e] champ de la table. Le premier synonyme de e1 est en 27 (e4), le suivant est en 26 (e2) et c'est le dernier. La table de débordement peut être pleine alors qu'il reste de la place en table principale. En cas de retrait de clé, la zone de débordement peut être gérée en liste libre (voir § 2.9.1.c, page 87).

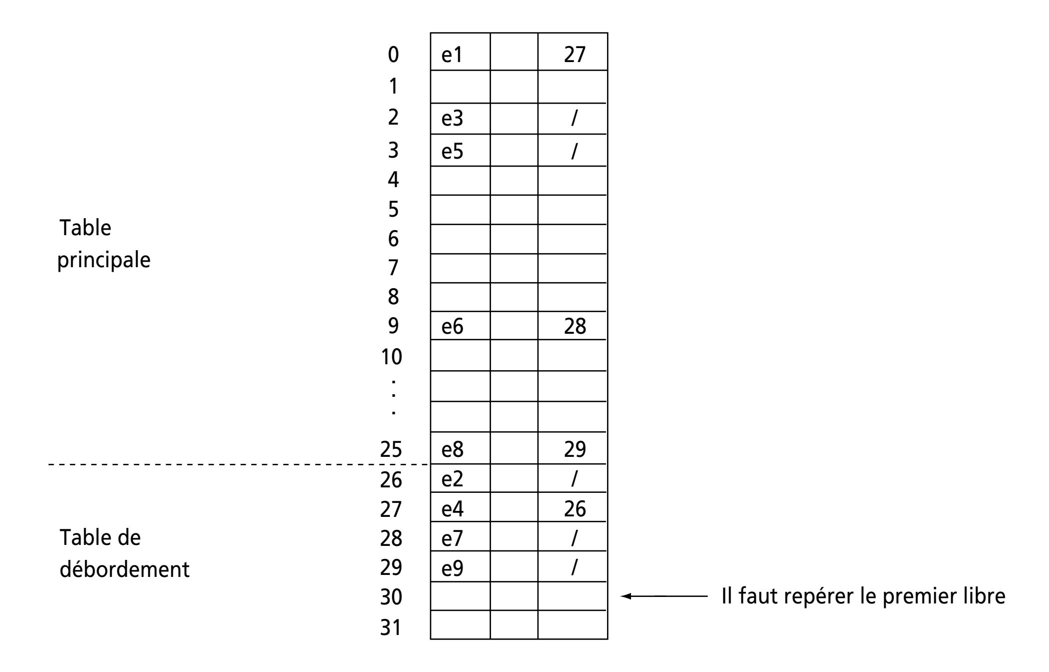


Figure 128 Chaînage en zone de débordement des synonymes.

4.3.12 Résolution par chaînage avec une seule table

4.3.12.a Expulsion de l'intrus

Devant la difficulté de gérer la table principale et la table de débordement, on peut décider de ranger les éléments dans la seule table principale qui a plus d'entrées que d'éléments à ranger. Il y a donc forcément des places disponibles. L'algorithme de rangement d'un élément x est le suivant :

– si h(x) est libre, ranger x en h(x).
– si h(x) est occupée par un élément x' tel que h(x) = h(x'),
  on cherche une entrée libre pour x et on établit le chaînage.

– si h(x) est occupée par un élément x' tel que h(x') est différent de h(x), x' est un intrus et doit être expulsé ailleurs :
  – on enlève x' de la liste,
  – on range x,
  – puis on cherche une nouvelle entrée libre pour x'.

Exemple :
Ranger les éléments suivants ; le chiffre entre parenthèses indique le hash-code : $e_1(1)$, $e_2(1)$, $e_3(3)$, $e_4(1)$, $e_5(2)$ pour une table de longueur N = 16. La résolution est pseudo-aléatoire (séquence 1, 6, 15, etc. pour N=16) avec chaînage des synonymes et expulsion des éléments occupant une entrée ne correspondant pas à leur hash-code.

e1 est rangé en 1, entrée libre.

pour e_2, l'entrée h(e2)=1 est occupée par e_1 tel que $h(e_1) = h(e_2)$; e2 est un synonyme de e1. On cherche une place pour e_2 : (1 +1) mod $_{16}$ soit 2 qui est libre. e2 est inséré dans l'entrée 2 ; on insère 2 (chaînage) en tête des éléments ayant hash-code 1.

e3 est rangé en 3, entrée libre.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	e1	e2	e3												
	1	1	3												
	2	-1	-1												

pour e_4, l'entrée h(e4)=1 est occupée par e_1 tel que $h(e_1) = h(e_4)$, e4 est un synonyme de e1, on cherche une place pour e_4 :

(1 + 1) mod $_{16}$ soit 2, déjà occupé

(1 + 6) mod $_{16}$ soit 7, libre

e4 est inséré dans l'entrée 7 ; on insère 7 (chaînage) en tête des éléments ayant hash-code 1.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	e1	e2	e3				e4								
	1	1	3				1								
	7	-1	-1				2								

pour $\mathbf{e_5}$, l'entrée h(e5) = 2 est occupée par un intrus e2 n'ayant pas 2 pour hash-code (h(e2) vaut 1) ;

– il faut déloger e_2 qui n'est pas tête de liste, en l'enlevant de sa liste,
– insérer e_5 qui est prioritaire à sa place,
– trouver une nouvelle place pour e_2

	h(x) + r (i)		mod 16		
i = 1	1 + 1	mod 16	soit	2	occupé
i = 2	1 + 6	mod 16	soit	7	occupé
i= 3	1 + 15	mod 16	soit	0	libre pour e2 qui est inséré en tête des éléments ayant hash-code 1.

La situation finale de la table après retrait de e2, insertion de e5, et réinsertion de e2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e2	e1	e5	e3				e4								
1	1	2	3				1								
7	0	-1	-1				-1								

Nombre d'accès à la table pour retrouver un élément (on suit le chaînage) :
1 : e1, e3, e5; 2 : e2; 3: e4.

4.3.12.b Cohabitation avec l'intrus

On peut aussi décider de ne pas expulser l'intrus n'ayant pas le même hash-code. On a alors des listes avec des éléments ayant des hash-codes différents, ce qui facilite l'insertion mais allonge la recherche dans la liste des synonymes.

Exemple

Ranger : $e_1(1)$, $e_2(1)$, $e_3(3)$, $e_4(1)$, $e_5(2)$

Comme précédemment, la résolution est pseudo-aléatoire (soit la séquence 1, 6, 9, 15, etc.) dans une table de longueur N = 16 avec chaînage et cohabitation (ou coalition). Les rangements des éléments e1, e2, e3, e4 conduisent à la situation suivante qui est la même que précédemment après insertion de e4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	e1	e2	e3				e4								
	1	1	3				1								
	7	-1	-1				2								

pour $\mathbf{e_5}$, h(e5) = 2, l'entrée 2 est occupée par e_2 tel que $h(e_2)$ est différent de $h(e_5)$.
on ne déplace pas $e_{2,}$ (cohabitation ou coalition)
on cherche une place pour e_5

i = 1	2 + 1	mod 16	soit	3	occupé
i = 2	2 + 6	mod 16	soit	8	libre pour e5

On insère e5 en tête de la liste commençant en $h(e_5)$ soit 2.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	e1	e2	e3				e4	e5							
	1	1	3				1	2							
	7	8	-1				2	-1							

On a donc, en entrée 1, une tête de liste qui contient des éléments ayant des hash-codes différents (d'où coalition) :

Liste des éléments en partant de l'entrée 1 : $e_1(1)$ - $e_4(1)$ - $e_2(1)$ - $e_5(2)$

Liste des éléments en partant de l'entrée 2 : $e_2(1)$ - $e_5(2)$

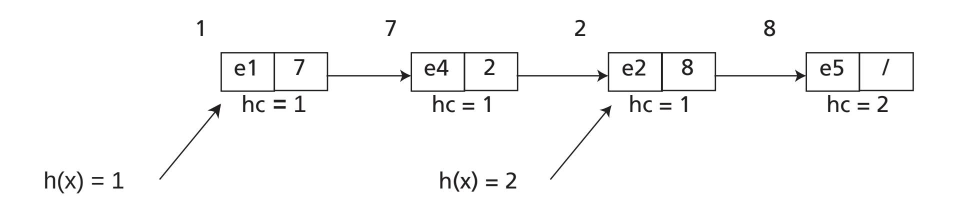


Figure 129 Chaînage avec coalition.

Nombre d'accès à la table pour retrouver un élément :

1 : e1, e3; 2 : e4, e5; 3 : e2.

4.3.12.c Le type TableHCC (table de hachage avec chaînage)

Le type *TableHC* décrit dans le fichier tablehc.h (voir § 4.3.8, page 228) doit être complété d'un champ entier *suivant* pour chaque élément de table. Ceci définit le type TableHCC (table de hachage avec chaînage) :

```
/* tablehcc.h  table de hash-code avec chainage */

#ifndef TABLEHCC_H
#define TABLEHCC_H

#include "fnhc.h"

#define NILE -1
typedef void Objet;

typedef struct {
  Objet* objet;
  int    suivant;  // chainage des synonymes
} ElementTable;

typedef struct {
  int nMax;          // nombre max (longueur) de la table
  int n;             // nombre d'éléments dans la table
  ElementTable* element;
  char* (*toString) (Objet*);
```

```
  int   (*comparer) (Objet*, Objet*);
  int   (*hashcode) (Objet*, int);
  int   (*resolution) (int, int, int);
} TableHCC;


TableHCC* creerTableHCC     (int nMax, char* (*toString) (Objet*),
                             int (*comparer) (Objet*, Objet*),
                             int (*hashcode) (Objet*, int),
                             int (*resolution) (int, int, int));
TableHCC* creerTableHCC     (int nMax);
booleen   insererDsTable    (TableHCC* table, Objet* nouveau);
Objet*    rechercherTable   (TableHCC* table, Objet* objetCherche);
void      listerTable       (TableHCC* table);
int       nbAcces           (TableHCC* table, Objet* objetCherche);
double    nbMoyAcces        (TableHCC* table);
void      listerEntree      (TableHCC* table, int entree);
void      ordreResolution   (TableHCC* table, int entree);

#endif
```

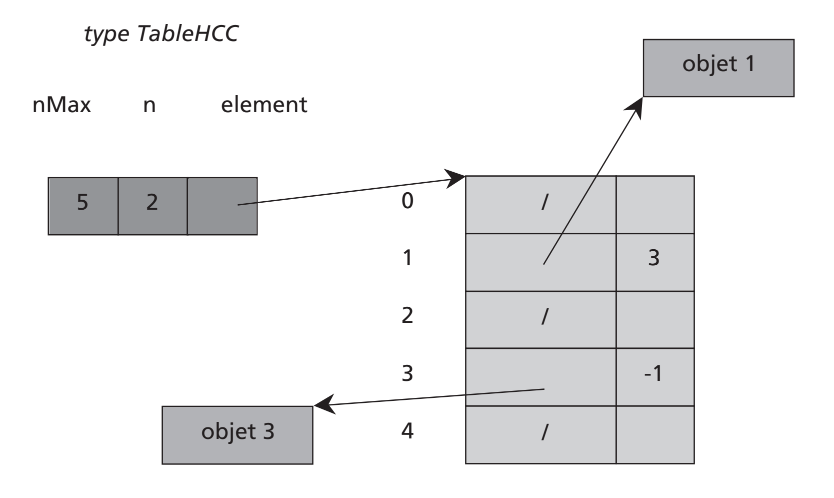


Figure 130 Table de hachage avec chaînage des synonymes (objet1 et objet3) de hc=1.

Les fonctions *creerTableHCC()*, *insererDsTable()*, *rechercherTable()*, etc., doivent être réécrites pour tenir compte du chaînage des synonymes.

```
TableHCC* creerTableHCC (int nMax, char* (*toString) (Objet*),
                                        int (*comparer) (Objet*, Objet*),
                                           int (*hashcode) (Objet*, int),
                                     int (*resolution) (int, int, int)) {
  TableHC* table    = new TableHCC();
```

```
  table->nMax       = nMax;
  table->n          = 0;
  table->element    = (ElementTable*) malloc (sizeof(ElementTable)
                                                          * nMax);
  table->toString   = toString;
  table->comparer   = comparer;
  table->hashcode   = hashcode;
  table->resolution = resolution;
  for (int i=0; i<nMax; i++) {
    table->element[i].objet   = NULL;  // fait par défaut
    table->element[i].suivant = NILE;  // -1 indique "pas de suivant"
  }
  return table;
}

TableHC* creerTableHCC (int nMax) {
  return creerTableHCC (nMax, toChar, comparerCar, hash1, resolution1);
}
```

La recherche se fait en suivant le chaînage des synonymes :

```
Objet* rechercherTable (TableHC* table, Objet* objetCherche) {
  booleen trouve = faux;
  int hc = table->hashcode (objetCherche, table->nMax);
  int re = hc;
  while (re!=NILE && !trouve) {
    printf ("rechercherTable entree re : %d\n", re);
    trouve = table->comparer (objetCherche, table->element[re].objet)
                                                            == 0;
    if (!trouve) re = table->element[re].suivant;
  }
  return re==-1 ? NULL : table->element[re].objet;
}
```

4.3.12.d Exemples de mise en œuvre du hachage avec chaînage

On reprend le programme et l'exemple du § 4.3.9, page 233, avec la fonction de hash-code hash1 et une résolution linéaire (résolution1) dans une table où les synonymes sont chaînés entre eux. Les trois éléments de hc=10 sont chaînés entre eux en 10, 13 et 11.

```
listerTable
  0 :
  1 :
  2 :
  3 :
  4 :
  5 : hc:  5 n:  1 svt: -1 Dufour Albert
  6 :
  7 :
  8 :
  9 :
 10 : hc: 10 n:  1 svt: 13 Punddo Jacques
 11 : hc: 10 n:  3 svt: -1 Dupond Jacques
 12 : hc: 12 n:  1 svt: -1 Duval Marie
 13 : hc: 10 n:  2 svt: 11 Ponddu Jacques
 14 :
 15 :
```

Exercice 25 - Hachage avec chaînage dans une seule table

Réécrire dans le cas du chaînage, les fonctions :

```
booleen insererDsTable (TableHCC* table, Objet* nouveau);
void    listerTable    (TableHCC* table);
int     nbAcces        (TableHCC* table, Objet* objetCherche);
```

Tester le menu et le programme principal de l'exercice 24, page 237. Les prototypes des fonctions sont les mêmes et ne modifient pas le programme de tests sauf pour les déclarations et les créations des tables.

4.3.13 Retrait d'un élément

Les méthodes de hachage sans chaînage des synonymes sont mal adaptées aux suppressions d'éléments. L'élément supprimé doit être marqué *détruit* et non *libre*, de façon à ne pas rompre l'accès aux éléments suivants. Il y a donc 3 états : occupé, libre et détruit. L'entrée d'un élément détruit pourra être réutilisée lors d'une nouvelle insertion.

4.3.14 Parcours séquentiel

L'accès séquentiel à tous les éléments de la table (ou du fichier) gérée suivant une méthode de hachage demande un test pour savoir si l'entrée est libre ou occupée. Le parcours séquentiel se fait dans l'ordre croissant des hash-codes. Si on veut un autre parcours (alphabétique par exemple), il faut faire un tri suivant la clé.

4.3.15 Évaluation du hachage

Les méthodes de hachage sont des méthodes qui permettent un accès rapide à partir d'une clé, les données se trouvant dans une table en mémoire centrale, ou dans une table sur disque (fichier en accès direct). On peut faire une évaluation mathématique du nombre moyen d'accès à la table pour retrouver un élément. On suppose que toutes les entrées peuvent être sollicitées de manière **équirépartie**. Le nombre moyen d'accès dépend du taux d'occupation de la table soit le rapport entre le nombre d'entrées occupées sur le nombre d'entrées réservées. A priori, au départ, le nombre d'entrées réservées doit être environ de deux fois le nombre d'éléments à mémoriser. Ce taux moyen ne dépend pas du nombre d'éléments dans la table ou fichier, ce qui en fait une excellente méthode pour accéder à de grands ensembles de données pour peu que l'on accepte de perdre de la place.

La Figure 131 donne les nombres moyens d'accès en fonction du taux d'occupation pour différentes méthodes. La méthode de résolution des collisions qui consiste à placer le synonyme sur les entrées qui suivent est la plus simple à programmer, mais également la moins performante lorsque le taux d'occupation de la table augmente. Les techniques de résolution par chaînage donnent d'excellents résultats

(on suit la liste des synonymes pour la recherche) au prix d'un encombrement légèrement supérieur puisqu'il faut mémoriser les chaînages.

A = taux d'occupation	linéaire r(i) = i	pseudo-aléatoire	chaînage
0.5	1.5	1.39	1.25
0.75	2.5	1.83	1.38
0.90	5.5	2.56	1.45

Figure 131 Nombre moyen d'accès en fonction du taux d'occupation[1].

Si on compare avec la recherche séquentielle ou même la recherche dichotomique, on voit que cette méthode donne d'excellents résultats. Pour un fichier de un million d'éléments :

recherche séquentielle = n / 2 = 500 000 accès

recherche dichotomique = $\log_2$ n = 20 accès

hachage avec chaînage = 1.45 accès avec un taux d'occupation de 90% (si les entrées de la table sont équiréparties).

Conclusions sur le hachage

Avantages des tables gérées par hachage

Le nombre d'accès pour retrouver un élément ne dépend pas de la taille de la table mais uniquement du taux d'occupation de la table. L'accès est <u>très</u> rapide.

Inconvénients

La taille de la table doit être fixée a priori et supérieure au nombre d'éléments à traiter. L'accès séquentiel aux éléments, suivant un ordre croissant ou décroissant de la clé, nécessite un tri. Le retrait d'éléments peut présenter quelques difficultés sauf dans le cas du chaînage.

4.3.15 Exemple 1 : arbre n-aire de la Terre (en mémoire centrale)

On veut accélérer la recherche dans un arbre binaire non ordonné en mémoire centrale, en créant une table accédée par hachage qui fournit un pointeur sur un nœud à partir de sa clé (nom). L'arbre binaire est d'abord créé. La table de hachage est ensuite créée au début de la consultation par parcours de l'arbre binaire. Les interrogations sur l'arbre binaire permettent de trouver un nœud plus rapidement en consultant la table plutôt qu'en parcourant l'arbre. La fonction de hachage retenue est la fonction `hash1()` (voir § 4.3.6, page 227). L'arbre n-aire considéré à titre d'exemple est celui de la nomenclature de la Terre donnée au § 3.2.11.b, page 147. Les hash-codes correspondant aux différentes clés dans une table de longueur HCMAX=70 sont les suivants :

1. D'après Robert Morris, Communication of the ACM, volume 11, numéro 11, janvier 1968

2 : Bretagne 7 : Afrique 8 : Belgique 10 : Europe 19 : Amerique 32 : Inde 34 : Bourgogne Asie 39 : Chine Irak	47 : France 52 : Oceanie 53 : Niger 54 : Congo 56 : Japon 60 : Corse 66 : Terre 67 : Espagne Danemark

Figure 132 Hash-codes des éléments de terre.nai.

Il y a collision pour l'entrée 34 (Bourgogne et Asie), l'entrée 39 (Chine et Irak) et l'entrée 67 (Espagne et Danemark). La table de hachage est gérée comme indiqué sur la Figure 133. Chaque élément de la table contient un pointeur sur un nœud de l'arbre.

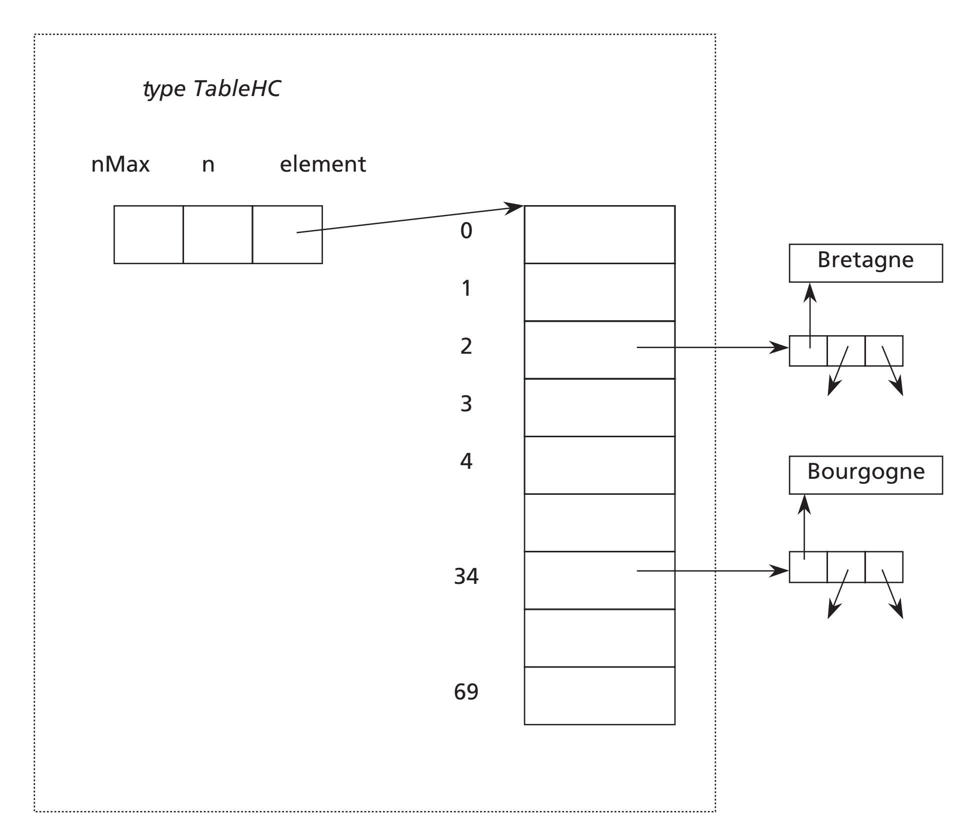


Figure 133 Table de hachage pour l'accès aux nœuds de l'arbre de la Terre.

Les déclarations sont les suivantes :

```
/* nomenclaturehc.cpp
   utilise le type Arbre et le type TableHC ou TableHCC */

#include <stdio.h>
#include <string.h>
```

```
#include "arbre.h"
#include "fnhc.h"

#ifndef CHAINAGE
#include "tablehc.h"
typedef TableHC Table;
#else
#include "tablehcc.h"
typedef TableHCC Table;
#endif

#define HCMAX 70
```

Pour chaque objet de la table, il faut fournir la référence du nœud.

```
char* toStringNd (Objet* objet) {
  Noeud* nd = (Noeud*) objet;
  return (char*) getobjet(nd);
}
```

Le hash-code d'un objet de la table s'obtient en utilisant la fonction *hashNd()*.

```
int hashNd (Objet* objet, int n) {
  Noeud* nd = (Noeud*) objet;
  return hash1( (char*) getobjet(nd), n);
}
```

La comparaison de deux objets de la table (deux pointeurs sur des nœuds de l'arbre) se fait en utilisant la fonction *comparer()*.

```
int comparer (Objet* objet1, Objet* objet2) {
  Noeud* nd1 = (Noeud*) objet1;
  Noeud* nd2 = (Noeud*) objet2;
  return strcmp ((char*) getobjet(nd1), (char*) getobjet(nd2));
}
```

La fonction **parcoursArbre**() parcourt l'arbre binaire pointé par *racine* et construit la table de hachage *table*. Pour chaque nœud visité, un pointeur sur ce nœud est ajouté dans la table à une place dépendant du hash-code du nœud et de la résolution.

```
void parcoursArbre (Noeud* racine, Table* table) {
  if (racine != NULL) {
     insererDsTable (table, racine);
     parcoursArbre (getsag(racine), table);
     parcoursArbre (getsad(racine), table);
  }
}
```

La fonction **construireTableHC**() construit la table à partir de l'arbre :

```
// initialiser et construire la table de hachage à partir de l'arbre
void construireTableHC (Arbre* arbre, Table* table) {
  parcoursArbre (getracine(arbre), table);
}
```

Le programme principal construit un arbre de caractères à partir du fichier terre.nai. Il construit ensuite une table de hachage (avec ou sans chaînage) par parcours de l'arbre. Il liste la table en indiquant les entrées occupées et effectue une recherche à l'aide de la table de hachage du nœud "France". Le sous-arbre du nœud "France" est alors dessiné.

```
void main () {
  printf ("Création d'un arbre binaire à partir d'un fichier\n");
  printf ("Donner le nom du fichier décrivant l'arbre n-aire ? ");
  char nomFE [50];
  //scanf ("%s", nomFE);
  strcpy (nomFE, "terre.nai");

  FILE*  fe = fopen (nomFE, "r");
  Arbre* arbre;
  if (fe == NULL) {
    printf ("%s erreur ouverture\n", nomFE);
  } else {
    arbre = creerArbreCar (fe);
  }
  dessinerArbreNAire (arbre, stdout);

  #ifndef CHAINAGE
  Table* table = creerTableHC (HCMAX, toStringNd, comparer,
                                                  hashNd, resolution1);
  #else
  Table* table = creerTableHCC (HCMAX, toStringNd, comparer,
                                                  hashNd, resolution1);
  #endif

  construireTableHC (arbre, table);
  listerTable (table);

  Noeud* objetCherche = cF ("France");
  Noeud* nd = (Noeud*) rechercherTable (table, objetCherche);
  if (nd != NULL) {
    printf ("trouvé %s\n", (char*) getobjet(nd));
    Arbre* arbre = creerArbre (nd);
    dessinerArbreNAire (arbre, stdout);
  } else {
    printf ("%s inconnu dans l'arbre\n", (char*) getobjet(objetCherche));
  }
}
```

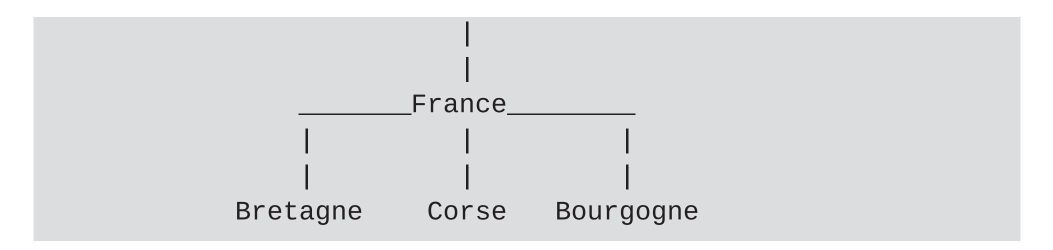
Exemple de résultats (seules les entrées non nulles de la table de hachage sont écrites).

```
 2 : hc:  2   1 Bretagne
 7 : hc:  7   1 Afrique
 8 : hc:  8   1 Belgique
10 : hc: 10   1 Europe
19 : hc: 19   1 Amerique
32 : hc: 32   1 Inde
34 : hc: 34   1 Bourgogne                      r(i) = i
35 : hc: 34   2 Asie
39 : hc: 39   1 Chine
```

```
        40 : hc: 39   2 Irak
        47 : hc: 47   1 France
        52 : hc: 52   1 Oceanie
        53 : hc: 53   1 Niger
        54 : hc: 54   1 Congo
        56 : hc: 56   1 Japon
        60 : hc: 60   1 Corse
        66 : hc: 66   1 Terre
        67 : hc: 67   1 Espagne
        68 : hc: 67   2 Danemark

       Nombre d'éléments dans la table : 19
       Taux d'occupation de   la table : 0.27
       Nombre moyen d'accès à la table : 1.16
```

Recherche à l'aide de la table de hachage du nœud France et dessin du sous-arbre.



```
                          |
                          |
                   _______France________
                  |         |         |
                  |         |         |
              Bretagne    Corse   Bourgogne
```

4.3.17 Exemple 2 : arbre n-aire du corps humain (fichier)

Soit l'arbre n-aire suivant du corps humain (voir exercice 17, page 142) :

```
homme:   tete cou tronc bras jambe;
tete:    crane yeux oreille cheveux bouche;
tronc:   abdomen thorax;
thorax:  coeur foie poumon;
jambe:   cuisse mollet pied;
pied:    cou-de-pied orteil;
bras:    epaule avant-bras main;
main:    doigt;
```

Si le nombre d'éléments décrivant l'arbre est important les éléments ne peuvent pas être gardés en mémoire centrale. Il faut donc les enregistrer dans un fichier. Les différents éléments de l'arbre binaire sont rangés dans un fichier en accès direct suivant une méthode de hachage. La fonction de hachage est la fonction *hash1()* (voir § 4.3.6, page 227) : *somme des rangs alphabétiques des lettres (blancs et tirets exclus) modulo 70.* La résolution des collisions se fait par chaînage en table de débordement (voir Figure 128).

Les hash-codes des éléments sont les suivants : tronc(0), yeux(5), oreille(6), cuisse(6), mollet(7), orteil(9), cou-de-pied(12), thorax(16), cheveux(18), poumon(24), avant-bras(28), jambe(31), pied(34), foie(35), main(37), cou(39), bras(40), crane(41), tete(50), homme(54), bouche(54), abdomen(54), doigt(55), epaule(60), coeur(62). Il y

a collision pour *oreille* et *cuisse* à l'entrée 6, et *homme*, *bouche*, *abdomen* qui prétendent tous les trois à l'entrée 54. L'arbre n-aire est mémorisé sous sa forme binaire en allocation contiguë (voir Figure 60), les numéros d'enregistrements occupés par les nœuds étant imposés par la fonction de hachage.

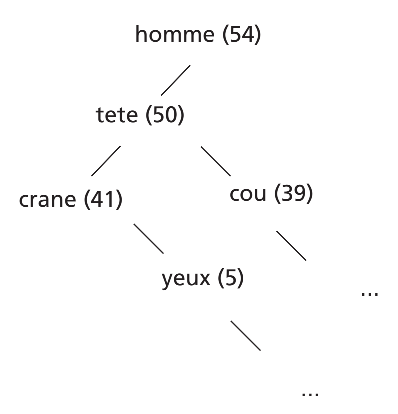


Figure 134 Dessin partiel de l'arbre et des places attribuées à chaque élément.

L'implantation de l'arbre en utilisant le hachage est indiquée ci-dessous. Seuls les enregistrements occupés sont affichés. La première colonne est un booléen qui indique si l'entrée est libre ou occupée. Les entrées 0, 5, 6, etc. sont occupées. Les entrées 1, 2, 3, 4, 8, etc. sont inoccupées. C'est une concession à faire à cette méthode : on utilise plus de places que nécessaire. La colonne 2 contient le nom du nœud ; c'est la clé de la fonction de hachage. La colonne (3) contient le pointeur sur le sous-arbre gauche (SAG), la colonne (4) le pointeur sur le sous-arbre droit (SAD). Ainsi, *tête* (hash-code 50) est rangé dans l'entrée 50 ; son SAG commence en 41 et son SAD en 39. La colonne (5) contient le chaînage des synonymes en zone de débordement. Il y a un synonyme pour *oreille* à l'entrée 6 ; ce synonyme *cuisse* est rangé en 72. Pour *homme*, il y a un synonyme en 71 (*abdomen*), suivi d'un autre synonyme en 70 (*bouche*).

	(1)	(2)	(3)	(4)	(5)
0	1	tronc	71	40	-1
5	1	yeux	-1	6	-1
6	1	oreille	-1	18	72
7	1	mollet	-1	34	-1
9	1	orteil	-1	-1	-1
12	1	cou-de-pied	-1	9	-1
16	1	thorax	62	-1	-1
18	1	cheveux	-1	70	-1
24	1	poumon	-1	-1	-1
28	1	avant-bras	-1	37	-1
31	1	jambe	72	-1	-1

```
 34  1                pied  12  -1  -1
 35  1                foie  -1  24  -1
 37  1                main  55  -1  -1
 39  1                 cou  -1   0  -1
 40  1                bras  60  31  -1
 41  1               crane  -1   5  -1
 50  1                tete  41  39  -1
 54  1               homme  50  -1  71
 55  1               doigt  -1  -1  -1
 60  1              epaule  -1  28  -1
 62  1               coeur  -1  35  -1
 70  1              bouche  -1  -1  -1
 71  1             abdomen  -1  16  70
 72  1              cuisse  -1   7  -1
```

La fonction *trouverNoeud()* (voir § 3.2.4.e, page 119) de recherche d'un nœud dans un arbre non ordonné en mémoire centrale, peut être améliorée en utilisant la fonction *PNoeud trouverNoeud (char* nomC, Noeud* enr) ;* définie ci-dessous qui utilise l'accès direct du hachage. On calcule le hash-code hc de *nomC* (nom cherché). Si l'entrée *hc* est inoccupée, *nomC* n'existe pas dans le fichier. Si l'entrée *hc* contient *nomC*, on a trouvé, sinon, il faut parcourir la liste des synonymes pour trouver *nomC*, ou pour conclure que *nomC* n'existe pas dans le fichier. La fonction *void lireD (int n, Noeud* enr) ;* effectue la lecture directe de l'enregistrement *n* qui est mémorisé, au retour de l'appel, à l'adresse contenue dans *enr*.

```
#define  NILE   -1

#define  MAXENR 100
#define  NBENTR 70

typedef  char   Chaine [16];
typedef  int    PNoeud;

typedef struct {
  booleen occupe;
  Chaine  nom;
  PNoeud  gauche;
  PNoeud  droite;
  PNoeud  syn;
} Noeud;

// lire directement l'enregistrement n,
// et le ranger à l'adresse pointée par enr
void lireD (int n, Noeud* enr) {
  fseek (fr, (long) n*sizeof (Noeud), 0);
  fread (enr, sizeof (Noeud), 1, fr);
}

// fournit le numéro de l'enregistrement contenant nomC
// si nomC existe, NILE sinon;
// *enr  contient l'enregistrement s'il a été trouvé
```

```
PNoeud trouverNoeud (char* nomC, Noeud* enr) {
  PNoeud pnom;
  PNoeud hc = hashCode (nomC, NBENTR);
  lireD (hc, enr);
  if (!enr->occupe) {
    pnom = NILE;
  } else if (strcmp (enr->nom, nomC) == 0) {
    pnom = hc;
  } else {
    pnom = NILE;
    PNoeud svt = enr->syn;
    booleen trouve = faux;
    while ( (svt!=NILE) && !trouve) {
      lireD (svt, enr);
      if ( strcmp (enr->nom, nomC) == 0) {
        pnom   = svt;
        trouve = vrai;
      } else {
        svt = enr->syn;
      }
    }
  }
  return pnom;
}
```

On peut bien sûr reprendre le menu concernant les parcours et les interrogations des arbres n-aires mémorisés sous forme binaire en mémoire centrale (voir § 3.2.11.a page 142). Le fait que l'arbre soit mémorisé dans un fichier ne change pas les algorithmes, seulement le codage (voir § 3.2.13, page 153).

```
Nom du Noeud dont on cherche les descendants n-aire ? thorax

Descendants n-aires de thorax
  coeur
  foie
  poumon
```

```
Nom du Noeud dont on cherche les ascendants n-aire ? orteil

Ascendants n-aires de orteil
  orteil
  pied
  jambe
  homme
```

Exercice 26 - Hachage sur l'arbre de la Terre

En utilisant la même fonction de hachage et la même méthode de résolution que sur l'exemple 2 du corps humain, donner le schéma d'implantation correspondant à l'arbre n-aire de la Terre décrit dans le § 3.2.11.b, page 147.

Exercice 27 - Table des étudiants gérée par hachage avec chaînage

- Créer un fichier *etudiant.dat* d'une centaine de noms (clés) différents (plus des informations spécifiques de chaque étudiant comme son prénom et son groupe

par exemple). En reprenant les algorithmes du cours, écrire un programme de création d'une table de hachage comprenant 197 entrées (de 0 à 196). La fonction de hachage est hash2() (somme des caractères ascii) et la résolution est du type r(i) = K * i (K=19) avec chaînage et expulsion de l'intrus. 19 et 197 sont premiers entre eux.

- Effectuer des recherches à partir des noms des étudiants.
- Sur l'exemple du fichier *etudiant.dat*, écrire le hash-code et le nombre d'accès pour chaque élément de la table, le taux d'occupation et le nombre moyen d'accès. Comparer à l'évaluation du cours (Figure 131, page 245).

4.4 RÉSUMÉ

Les tables sont des structures de données permettant de mémoriser des ensembles de valeurs et leurs attributs, et de retrouver (le plus rapidement possible) les différents attributs à partir de la clé d'un élément (son nom par exemple). Lorsque le nombre d'éléments de la table est faible (inférieur à une centaine) ou que les recherches dans la table sont peu fréquentes, on peut envisager une recherche séquentielle qui est simple à mettre en œuvre. Si le nombre d'éléments est faible, mais que les recherches sont très fréquentes, on peut optimiser l'algorithme de recherche séquentielle en utilisant la méthode de la sentinelle. Si insertions et recherches se font en deux phases séparées, on peut ordonner la table en cours ou en fin d'insertion, et effectuer par la suite, en mémoire centrale, des recherches dichotomiques. Si le nombre d'éléments est important sur mémoire secondaire, il convient de limiter les accès disque en regroupant les clés dans des sous-tables qui sont amenées en mémoire centrale en un seul accès disque, la recherche se poursuivant en mémoire centrale dans cette sous-table. Ce partitionnement en sous-tables peut utiliser les techniques des B-arbres s'il y a des ajouts et retraits d'éléments.

Dans certains cas, lorsque la clé est structurée, on peut effectuer un calcul à partir de cette clé qui fournit la place dans la table (ou fichier) de cette clé. Dans le cas général, une autre technique très performante consiste à définir une fonction qui fournit également la place dans la table à partir de la clé. Cependant, cette fonction peut fournir une même place pour deux clés différentes ; il faut donc en cas de conflit trouver une autre place pour l'élément synonyme (en collision). Ces techniques de hachage sont très rapides et indépendantes du nombre d'éléments dans la table. Les performances dépendent seulement du taux d'occupation de la table qui doit être surdimensionnée ; on réserve plus de places que strictement nécessaire. Par contre, le traitement séquentiel suivant l'ordre de la clé nécessite une copie et un tri de la table, ce qui n'est pas gênant si ce traitement est peu fréquent.

Chapitre **5**

Les graphes

5.1 DÉFINITIONS

Un graphe est une structure de données composée d'un **ensemble de sommets**, et d'un **ensemble de relations entre ces sommets**.

Si la relation n'est pas orientée, la relation est supposée exister dans les deux sens. Le graphe est dit non orienté ou symétrique. Dans le cas contraire, si les relations sont orientées, le graphe est dit orienté. Une relation est appelée un **arc** (quelquefois une arête pour les graphes non orientés). Les sommets sont aussi appelés nœuds ou points.

5.1.1 Graphes non orientés (ou symétriques)

Le graphe de la Figure 135 peut se noter comme suit :

- S = {S1, S2, S3, S4, S5} ; ensemble des sommets
- A = {S1S2, S1S3, S2S3, S3S4, S3S5, S4S5} ; ensemble des relations symétriques. Par exemple, S1S2 est vrai, de même que S2S1.

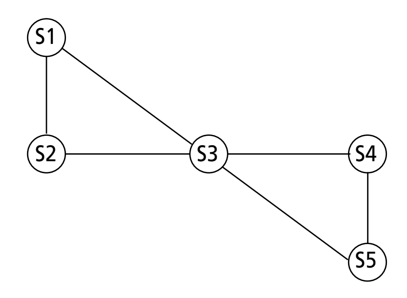


Figure 135 Un graphe non orienté : les relations existent dans les deux sens.

Graphe connexe : un graphe non orienté est dit connexe si on peut aller de tout sommet vers tous les autres sommets. Le graphe de la Figure 135 est connexe ; celui de la Figure 136 ne l'est pas ; il est constitué de deux composantes connexes.

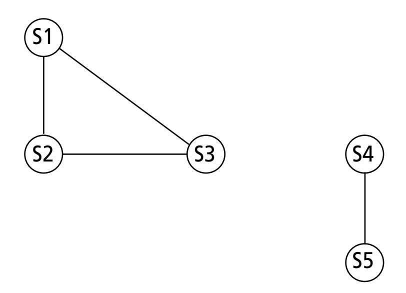


Figure 136 Un graphe non connexe et ses deux composantes connexes.

5.1.2 Graphes orientés

Si les relations sont orientées, le graphe est dit **orienté**.

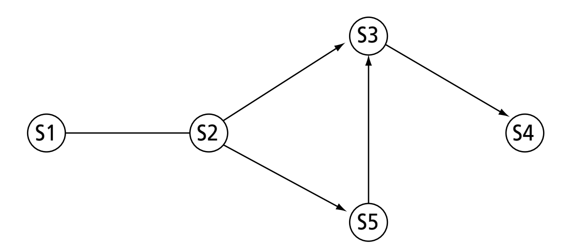


Figure 137 Un graphe orienté.

Pour un arc S1S2, S2 est dit successeur de S1 ou encore adjacent à S1 ; S1 est le prédécesseur de S2.

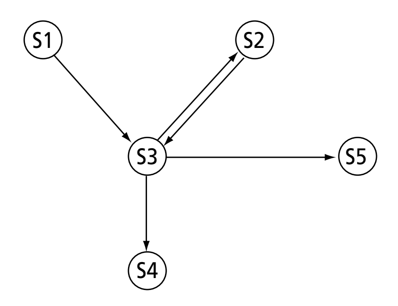


Figure 138 Degrés d'un graphe orienté.

d°(S3)	= 5	degré du sommet S3 : nombre d'arcs entrants ou sortants
d+(S3)	= 3	nombre d'arcs sortants : demi-degré extérieur ou degré d'émission
d-(S3)	= 2	nombre d'arcs entrants : demi-degré intérieur ou degré de réception

Graphe fortement connexe : un graphe orienté est dit fortement connexe si on peut aller de tout sommet vers tous les autres sommets (en passant éventuellement par un ou plusieurs sommets intermédiaires).



Figure 139 Graphe non fortement connexe et composantes fortement connexes.

5.1.3 Graphes orientés ou non orientés

Les définitions suivantes s'appliquent aux graphes orientés comme aux graphes non orientés.

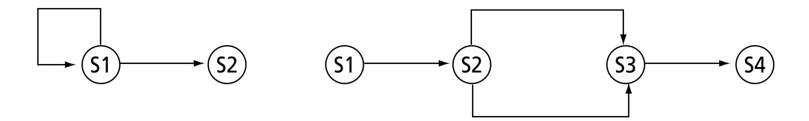


Figure 140 Une boucle et un graphe multiple.

Une boucle (autoboucle) est une relation (Si, Si). Un multigraphe ou graphe multiple est un graphe tel qu'il existe plusieurs arcs entre certains sommets. Sur la Figure 140, la relation S2S3 existe 2 fois ; le graphe est un 2-graphe orienté ou plus généralement un p-graphe. Un graphe **simple** est un graphe sans boucle et sans arc multiple.

Un graphe est dit valué (pondéré) si à chaque arc on associe une valeur représentant le coût de la transition de cet arc.

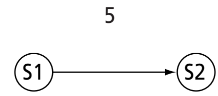


Figure 141 Un graphe valué : la transition S1 vers S2 coûte 5.

Un **chemin** dans un graphe est une suite d'arcs consécutifs.

La **longueur d'un chemin** est le nombre d'arcs constituant ce chemin.

Un **chemin simple** est un chemin où aucun arc n'est utilisé 2 fois.

Un **circuit simple** est un chemin simple tel que le premier et le dernier sommet sont les mêmes.

Un **circuit eulérien** est un circuit qui passe une et une seule fois par tous les arcs.

Un **circuit hamiltonien** est un circuit qui passe une et une seule fois par tous les sommets.

Un graphe est **planaire** si aucun de ses arcs ne se coupent. Il est impossible par exemple de relier 3 puits vers 3 maisons sans que les arcs ne se coupent, chaque puits étant relié à chacune des maisons.

> ***Remarques*** : dans les graphes non orientés, on parle quelquefois de chaîne au lieu de chemin et de chaîne simple pour un chemin simple. Le terme arête est aussi utilisé à la place d'arc. Le terme de cycle indique un circuit simple orienté.

Un graphe non orienté peut toujours être considéré comme un graphe orienté où les relations symétriques sont explicitement mentionnées.

5.2 EXEMPLES DE GRAPHES

Exemple 1 : Un réseau de communication (routier, aérien, électrique, d'alimentation en eau, etc.) entre différents lieux peut être schématisé sous forme d'un graphe comme l'indique la Figure 142. Le lieu peut être une ville ; il peut aussi indiquer différents carrefours ou places dans une ville. Le graphe est symétrique : les relations existent dans les deux sens. On peut aller par exemple de Rennes vers Nantes ou de Nantes vers Rennes. Ceci ne serait pas vrai s'il y avait par exemple des sens interdits (cas de la circulation dans une ville).

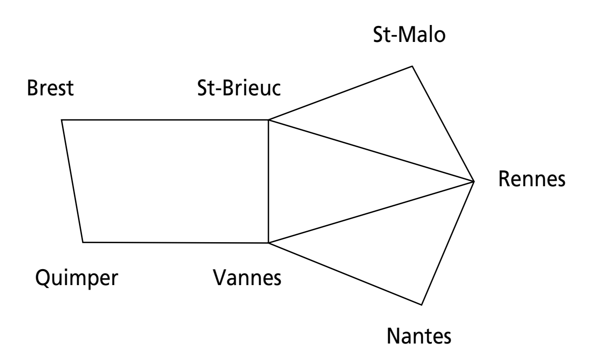


Figure 142 Un réseau de communication.

Exemple 2 : ordonnancement de tâches (graphe orienté sans cycle)
La construction d'un complexe immobilier, ou plus simplement d'une maison, d'un appareil compliqué (fusée par exemple) s'effectue en suivant un ordre bien précis dans l'accomplissement des différents travaux. Pour commencer certains travaux, il faut que d'autres soient terminés. Certains travaux peuvent cependant se réaliser en parallèle. Le graphe peut être valué, et certains travaux sont dits critiques, car si on prend du retard pour ceux-ci, le projet en entier sera retardé. Pour des travaux en parallèle, le plus long chemin est critique ; pour les autres, il y a une certaine latitude qui ne retarde pas le projet.

Le graphe présenté Figure 143 est un graphe d'ordonnancement non valué. Dans un but pédagogique, on conseille d'étudier les différentes notions dans l'ordre indiqué par le graphe, pour finalement aboutir au diplôme. En fait, comme les notions sont très imbriquées, il est difficile d'établir un ordre séquentiel des cours. Le graphe est orienté sans cycle

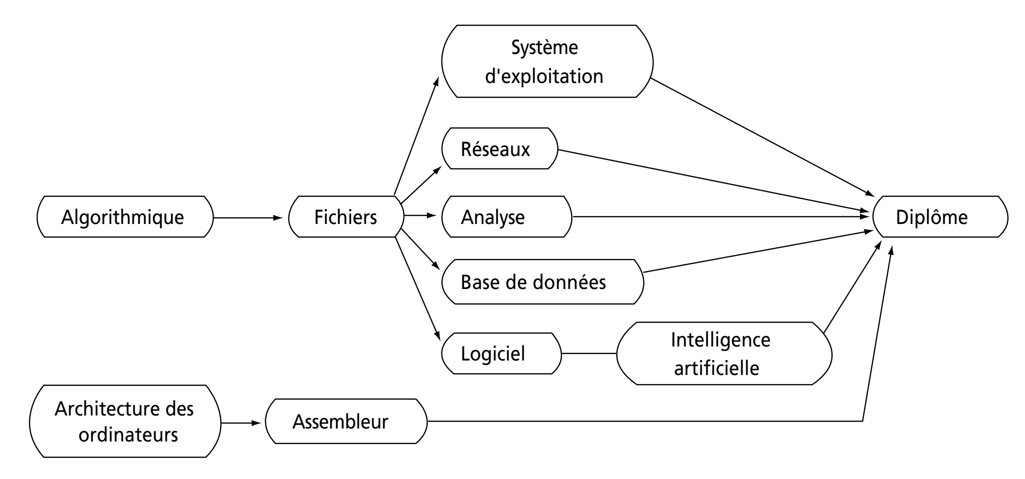


Figure 143 Un graphe d'ordonnancement.

Exemple 3 : un labyrinthe
Un labyrinthe peut être représenté par un graphe comme l'indique la Figure 144. L'entrée se fait en A, la sortie en H. Chaque carrefour présentant un choix de chemins est un sommet.

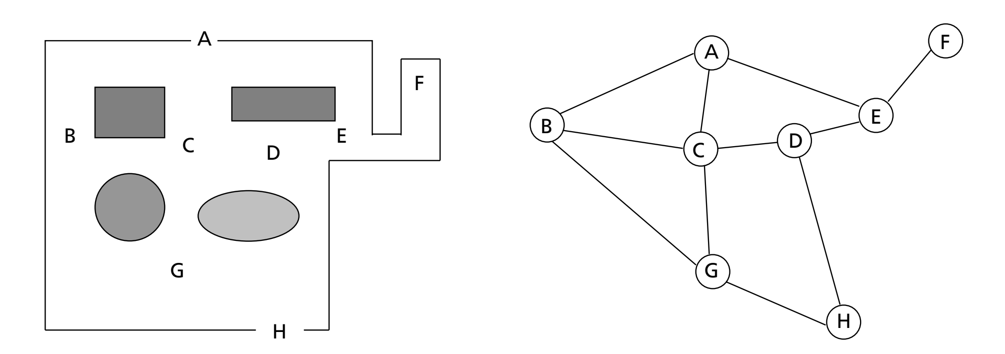


Figure 144 Un labyrinthe et son graphe équivalent

Exemple 4 : un programme peut être considéré comme un graphe orienté. Les sommets représentent les actions ; les arcs représentent l'enchaînement des actions.

Exemple : programme de calcul de la factorielle de n (voir *factorielleIter* § 1.2.1, page 4) :

```
int f, i;

f = 1;
for (i=1; i<=n; i++) {
  f = f * i;
}
printf (“%d”, f);
```

La schématisation est donnée sous forme d’organigramme et sous forme de graphe. Le graphe est orienté avec cycle.

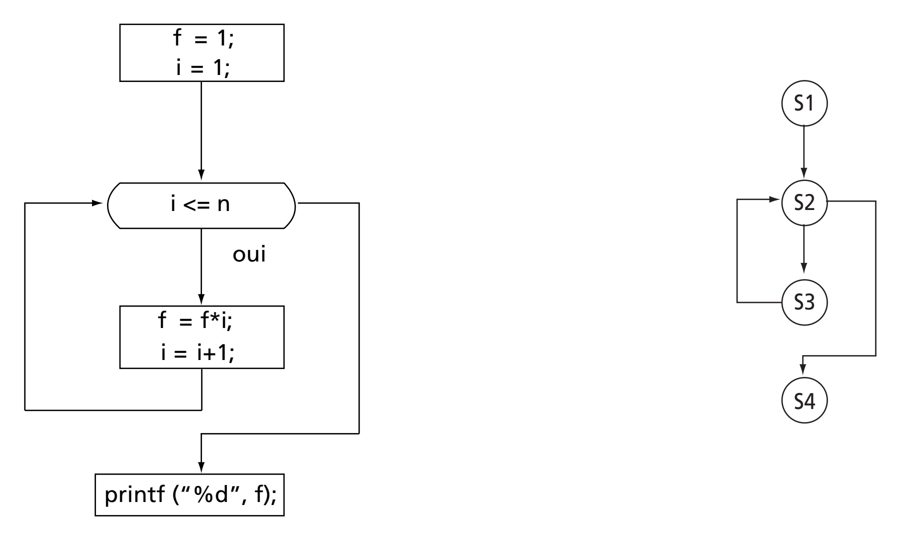


Figure 145 Programme schématisé sous la forme d’un graphe.

5.3 MÉMORISATION DES GRAPHES

Suivant le rapport entre le nombre de sommets et le nombre d’arcs, on choisit soit une mémorisation sous forme de matrice (rapport important), soit une mémorisation sous forme de listes d’adjacence. Dans ce dernier cas, la matrice serait dite creuse avec beaucoup d’éléments mémorisés inutilement.

5.3.1 Mémorisation sous forme de matrices d’adjacence

Chaque arc (i, j) est représenté par un booléen V (vrai) dans la matrice. Une valeur F (faux) non notée sur le schéma de la Figure 146 indique une absence de relation entre le sommet i et le sommet j. Le tableau nomS contient les noms des sommets et leurs caractéristiques. On peut facilement ajouter des sommets (si ns : nombre de sommets < nMax : nombre maximum de sommets) et des arcs entre deux sommets à partir de leur nom.

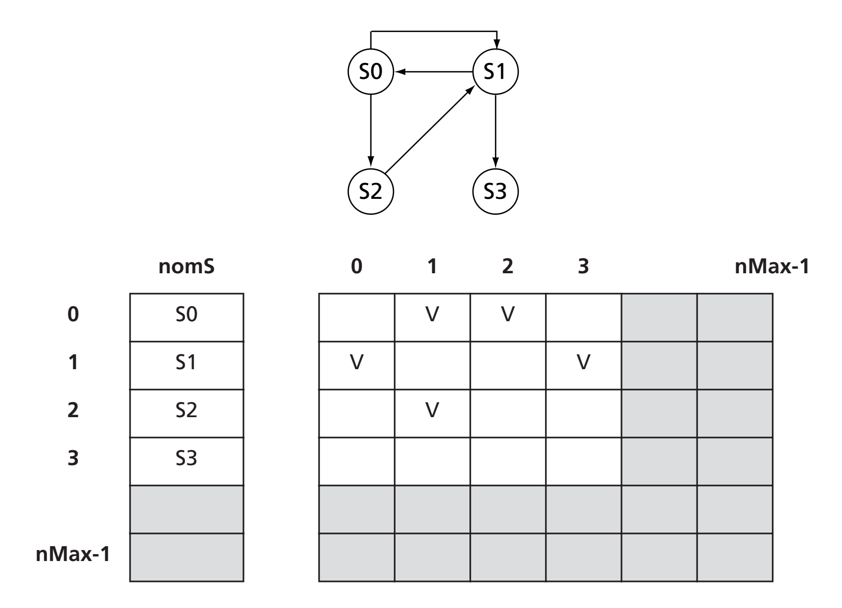


Figure 146 Mémorisation sous forme d'une matrice d'adjacence.

5.3.2 Mémorisation en table de listes d'adjacence

La partie concernant les caractéristiques des sommets est mémorisée dans une table (voir Chapitre 4) contenant, pour chaque entrée, une liste des sommets que l'on peut atteindre directement en partant du sommet correspondant à cette entrée. Par exemple, du sommet 0 (S0), on peut aller au sommet numéro 1 (S1) et au sommet numéro 2 (S2). On pourrait aussi mémoriser les listes dans un tableau en allocation contiguë (voir Figure 44, page 88).

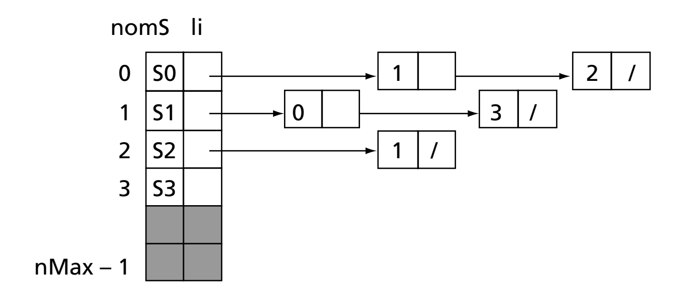


Figure 147 Mémorisation sous forme d'une table de listes d'adjacence.

5.3.3 Liste des sommets et listes d'adjacence : allocation dynamique

On peut tout allouer dynamiquement : la liste des sommets, et pour chaque sommet, la liste des sommets successeurs. On n'a plus besoin de définir nMax ; l'allocation est entièrement dynamique. Le premier champ des listes de successeurs

contient soit le numéro ou le nom du sommet successeur, soit un pointeur sur le sommet. Le dernier cas facilite l'accès au sommet sinon il faut parcourir la liste des sommets pour retrouver l'adresse du sommet et ses caractéristiques. Si le graphe est valué, il faut ajouter le poids de l'arc dans chacun des éléments des listes de successeurs.

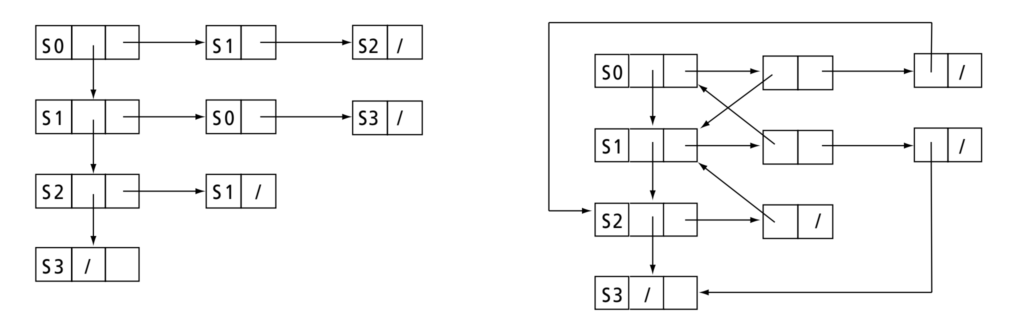


Figure 148 Variantes de la mémorisation sous forme de listes d'adjacence.

5.4 PARCOURS D'UN GRAPHE

Il s'agit d'écrire un algorithme qui permet d'examiner les sommets une et une seule fois. La présence de circuits doit être prise en considération de façon à ne pas visiter plusieurs fois le même sommet. Il faut donc marquer les sommets déjà visités. Comme pour les arbres (voir § 3.2.4, page 114), on distingue deux types de parcours : le parcours en profondeur et le parcours en largeur.

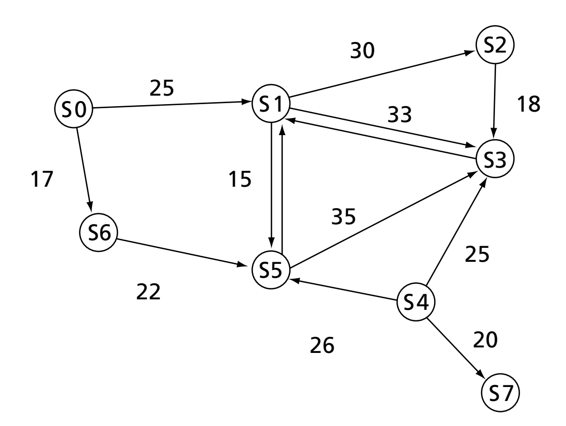


Figure 149 Un graphe valué de distances entre lieux.

Le graphe de la Figure 149 peut être décrit comme suit :

```
S0 S1 S2 S3 S4 S5 S6 S7 ;     liste des sommets
S0: S1 (25) S6 (17) ;         de S0, on peut aller en S1 et S6
```

```
S1: S2 (30) S3 (33) S5 (15) ;
S2: S3 (18) ;
S3: S1 (33) ;
S4: S3 (25) S5 (26) S7 (20) ;
S5: S1 (15) S3 (35) ;
S6: S5 (22) ;
```

5.4.1 Principe du parcours en profondeur d'un graphe

On part d'un sommet donné. On énumère le premier fils de ce sommet (par ordre alphabétique par exemple), puis on repart de ce dernier sommet pour atteindre le premier petit-fils, etc. Il s'agit pour chaque sommet visité, de choisir un des sommets successeurs du sommet en cours, jusqu'à arriver sur une impasse ou un sommet déjà visité. Dans ce cas, on revient en arrière pour repartir avec un des successeurs non visité du sommet courant. En partant de S0 sur la Figure 149, on peut aller en S1 ou S6. On choisit S1. De S1, on peut aller en S2, S3 ou S5. On choisit S2. De S2, on peut aller en S3, seule possibilité. De S3, on pourrait aller en S1 mais S1 a déjà été marqué. On revient en arrière sur S2 où il n'y a pas d'autre alternative. On revient en arrière sur S1 ; reste à essayer S3 et S5. S3 a déjà été visité. On prend donc le chemin S5. De S5, on ne peut explorer de nouveaux sommets. On revient en S1, puis S0. Pour S0, il reste une alternative vers S6.

Tous les sommets n'ont pas été visités en partant de S0. Il faut repartir d'un des sommets non encore visités, et essayer d'explorer en partant de ce sommet. On repart de S4 qui mène à S7.

L'indentation met en évidence le parcours en profondeur :

```
S0
    S1
        S2
            S3
        S5
    S6
S4
    S7
```

L'ordre de parcours en profondeur du graphe est donc le suivant : S0, S1, S2, S3, S5, S6, S4, S7.

5.4.2 Principe du parcours en largeur d'un graphe

On part d'un sommet donné. On énumère tous les fils (les suivants) de ce sommet, puis tous les petits-fils non encore énumérés, etc. C'est une énumération par génération : les successeurs directs, puis les successeurs au 2e degré, etc.

En partant de S0 sur la Figure 149, on visite S1 et S6. De S1, on visite S2, S3 et S5. De S6, on ne peut pas explorer de nouveaux sommets. De S2, S3 et S5, on ne peut pas explorer de nouveaux sommets. Il faut également repartir d'un sommet non

encore visité et non accessible de S0. On repart avec S4 qui nous conduit à S7. Le graphe entier a été parcouru.

Parcours en largeur sur l'exemple de la Figure 149 :

```
Parcours en largeur

 S0 S1 S6 S2 S3 S5
 S4 S7
```

L'ordre de parcours en largeur du graphe est donc le suivant : S0 S1 S6 S2 S3 S5 S4 S7.

5.5 MÉMORISATION (TABLE DE LISTES D'ADJACENCE)

La mémorisation peut se faire en utilisant la notion de table pour enregistrer les sommets et leurs caractéristiques. Le type Graphe utilise le type Table. Chaque objet de table est caractérisé par le nom du sommet (la clé), un booléen marqué qui est utilisé lors des parcours pour savoir si on est déjà passé par ce sommet et une liste li des sommets successeurs.

	element		
	clé	**marque**	**li**
0	**S0**		
1	**S1**		
2	**S2**		
3	**S3**		
4	**S4**		
5	**S5**		
6	**S6**		
7	**S7**		
n = 8			
nMax-1			

Figure 150 La partie table correspondant au graphe de la Figure 149. Le détail de l'implémentation est donné sur les figures suivantes.

La partie liste utilise le module de gestion des listes vu au chapitre 2 (voir § 2.3.8, page 48). Chaque élément de liste d'une entrée de la table contient un pointeur vers le sommet successeur et le coût de la relation (voir Figure 151). Du sommet S0, on peut aller en S1 (coût:25) ou en S6 (coût:17).

5.5.1 Le type Graphe

5.5.2 Le fichier d'en-tête des graphes

Le type Graphe peut donc être défini comme suit à partir du type Table et du type Liste.

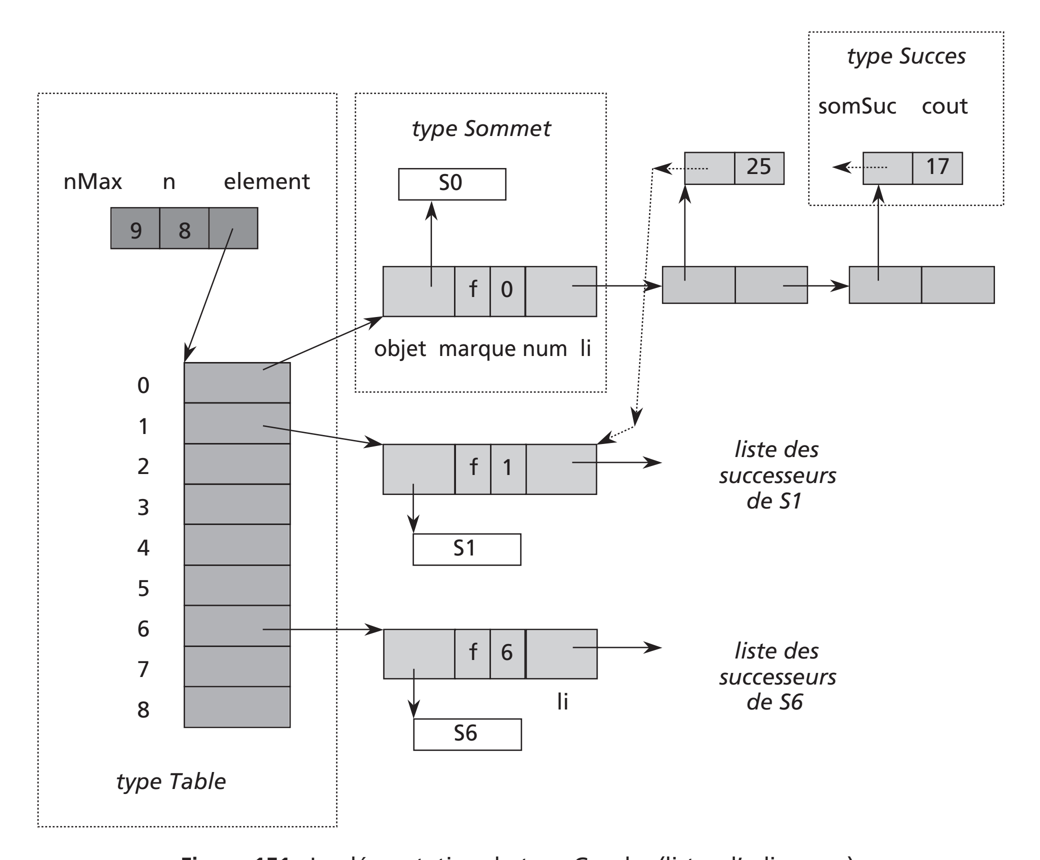


Figure 151 Implémentation du type Graphe (listes d'adjacence).

```
/* grapheadj.h  graphe avec des listes d'adjacence */

#ifndef GRAPHEADJ_H
#define GRAPHEADJ_H

#include "liste.h"
#include "table.h"
```

```
#define INFINI    INT_MAX

typedef struct {
  Objet*  objet;   // les caractéristiques du sommet (son nom)
  booleen marque;  // booléen marqué (pour le parcours)
  int     num;     // numéro du sommet dans la table
  Liste   li;      // liste des successeurs du sommet
} Sommet;

// successeur
typedef struct  {
  Sommet*  somSuc; // pointeur sur le sommet successeur
  int      cout;
} Succes;

typedef struct {
  Table*  table;   // la table représentant le graphe
  booleen value;   // le graphe est-il valué ?
} Graphe;

Graphe* creerGraphe    (int nMax, booleen value, char* (*toString) (Objet*),
                                 int (*comparer) (Objet*, Objet*));
Graphe* creerGraphe     (int nMax, int value);
void    detruireGraphe  (Graphe* graphe);
void    ajouterUnSommet (Graphe* graphe,  Objet* sommet);

void    ajouterUnArc    (Graphe* graphe,  Objet* sommetDepart,
                                          Objet* sommmetArrivee,
                                          int cout);
Graphe* lireGraphe      (FILE*   fe, int nMax);

void    ecrireGraphe    (Graphe* graphe);
void    parcoursProfond (Graphe* graphe);
void    parcoursLargeur (Graphe* graphe);
void    plusCourt       (Graphe* graphe, int nsi);

char* toStringSommetCar (Objet* objet);
int   comparerSommetCar (Objet* objet1, Objet* objet2);
#endif
```

5.5.3 Création et destruction d'un graphe

La fonction *Graphe* creerGraphe (int nMax, int value);* alloue dynamiquement une table de nMax entrées ; nMax est le nombre maximum de sommets envisagés dans la table. La fonction *razMarque()* met par défaut tous les champs *marque* des sommets à faux (sommet non visité). Les têtes de listes *li* de chaque entrée de la table sont initialisées (listes vides). *value* indique si le graphe est valué ou non.

```
/* grapheadj.cpp  module sur les graphes mémorisés
                  avec une table de listes d'adjacence */

#include "grapheadj.h"

// pointeur sur le nième sommet (élément de la table)
Sommet* getsommet (Graphe* graphe, int n) {
  return (Sommet*) graphe->table->element[n];
}
```

```
// marquer le sommet n (visité)
void marquersommet(Graphe* graphe, int n) {
  getsommet(graphe,n)->marque = vrai;
}

// le sommet n a-t'il été visité ?
booleen estmarque (Graphe* graphe, int n) {
  return getsommet(graphe,n)->marque;
}

// nom du sommet pointé par sommet
char* nomsommet (Graphe* graphe, Sommet* sommet) {
  return graphe->table->toString (sommet);
}

// nom du nième sommet
char* nomsommet (Graphe* graphe, int n) {
  return nomsommet (graphe, getsommet(graphe, n));
}

// nombre de sommets dans le graphe
int nbsommet (Graphe* graphe) {
  return graphe->table->n;
}

// remise à faux du tableau marqué (pour les parcours de graphe)
static void razMarque (Graphe* graphe) {
  for (int i=0; i<nbsommet(graphe); i++) getsommet(graphe,i)->marque = faux;
}

// créer et initialiser un graphe avec nMax sommets
Graphe* creerGraphe (int nMax, booleen value, char* (*toString) (Objet*),
                                         int (*comparer) (Objet*, Objet*)) {
  Graphe* graphe = new Graphe();
  graphe->table  = creerTable (nMax, toString, comparer);
  graphe->value  = value;
  return graphe;
}

// créer et initialiser un graphe avec nMax sommets
// par défaut, les sommets sont des chaînes de caractères
Graphe* creerGraphe (int nMax, booleen value) {
  return creerGraphe (nMax, value, toStringSommetCar, comparerSommetCar);
}
```

La fonction *detruireGraphe()* effectue le travail inverse de *creerGraphe()*, en désallouant les listes de chaque entrée *li* de la table, et en désallouant la table des sommets du graphe.

```
void detruireGraphe (Graphe* graphe) {
  Table* table = graphe->table;
  for (int i=0; i<table->n; i++) {
    Sommet* sommet = getsommet (graphe,i);
    detruireListe (&sommet->li);
  }
  detruireTable (table);
}
```

5.5.4 Insertion d'un sommet ou d'un arc dans un graphe

La fonction *ajouterUnSommet()* ajoute au graphe, le sommet objet. Cet ajout est sous-traité à *insererDsTable()* (voir § 4.1.3, page 200).

```
void ajouterUnSommet (Graphe* graphe,  Objet* objet) {
  Sommet* sommet = new Sommet();
  sommet->objet  = objet;
  sommet->marque = faux;
  Table* table   = graphe->table;
  sommet->num    = lgTable (table);  // le numéro du sommet
  initListe (&sommet->li);
  insererDsTable (graphe->table, sommet);
}
```

La fonction *ajouterUnArc()* ajoute au graphe, un arc entre les sommets de nom *somDepart* et *somArrivee*. La recherche du nom s'effectue grâce à la fonction *accesSequentiel()* de recherche séquentielle dans une table. Un élément de type Succes (successeur) est créé, rempli et inséré en fin de la liste des successeurs de *somDepart* (voir Figure 151).

```
// ajouter un arc entre deux objets (deux villes par exemple)
void ajouterUnArc (Graphe* graphe, Objet* sommetDepart,
                   Objet* sommetArrivee, int cout) {
  // rechercher un pointeur sur le sommet de départ
  Sommet d;
  d.objet = sommetDepart;
  Sommet* pSomD = (Sommet*) accesSequentiel (graphe->table, &d);
  if (pSomD == NULL) {
    printf ("Sommet %s inconnu\n", nomsommet(graphe, &d)); return;
  }

  // rechercher un pointeur sur le sommet d'arrivée
  Sommet a;
  a.objet = sommetArrivee;
  Sommet* pSomA = (Sommet*)accesSequentiel (graphe->table, &a);
  if (pSomA == NULL) {
    printf ("Sommet %s inconnu\n", nomsommet(graphe,&a)); return;
  }

  // enregistrer la relation entre les deux sommets
  Succes* succes = new Succes();
  succes->somSuc = pSomA;
  succes->cout   = cout;
  insererEnFinDeListe (&pSomD->li, succes);
}
```

5.5.5 Écriture d'un graphe (liste d'adjacence)

Écrire les sommets et les relations (arcs) d'un graphe :

```
void ecrireGraphe (Graphe* graphe) {
  printf ("\n\ngraphe %s\n\n", graphe->value ? "valué" : "non valué");
  for (int i=0; i<nbsommet(graphe); i++) {
    printf ("%s ", nomsommet (graphe,i));
  }
  printf (";\n");
```

```
  for (int i=0; i<nbsommet(graphe); i++) {
    Sommet* sommet = getsommet (graphe,i);
    Liste* li = &sommet->li;
    printf ("%s : ", nomsommet (graphe,i));
    ouvrirListe (li);
    while (!finListe (li) ) {
      Succes* succes = (Succes*) objetCourant (li);
      printf ("%s ", nomsommet (graphe, succes->somSuc));
      if (graphe->value) printf ("(%3d) ", succes->cout);
    }
    printf (";\n");
  }
}
```

5.5.6 Parcours en profondeur (listes d'adjacence)

Voir les explications en 5.4.1, page 262.

```
// Parcours récursif des successeurs de sommet; niveau permet
// de faire une indentation à chaque appel récursif
static void profondeur (Graphe* graphe, Sommet* sommet, int niveau) {
  sommet->marque = vrai;
  for (int i=1; i<niveau; i++) printf ("%5s"," ");
  printf ("%s\n", nomsommet (graphe, sommet));

  Liste* li = &sommet->li;
  ouvrirListe (li);
  while (!finListe (li) ) {
    Succes* succes = (Succes*) objetCourant (li);
    if (!succes->somSuc->marque)
      profondeur (graphe, succes->somSuc, niveau+1);
  }
}

// parcours en profondeur de graphe
void parcoursProfond (Graphe* graphe) {
  razMarque (graphe);
  for (int i=0; i<nbsommet(graphe); i++) {
    // sommet : pointeur sur le ième sommet de graphe
    Sommet* sommet = getsommet (graphe, i);
    if (!sommet->marque) profondeur (graphe, sommet, 1);
  }
}
```

5.5.7 Parcours en largeur (listes d'adjacence)

Comme pour les arbres (voir § 3.2.4.f, page 120), le parcours en largeur nécessite l'utilisation d'une liste (file d'attente) des sommets à traiter. *ajouterDsFile()* ajoute (un pointeur sur) un sommet en fin de la liste. On part d'un sommet non marqué (le premier par exemple), on l'insère dans la liste. On retire le premier élément de la liste en le remplaçant par ses successeurs non encore marqués jusqu'à ce que la liste soit vide. S'il reste un sommet non marqué, on recommence avec ce sommet.

```
// ajouter sommet dans la file
static void ajouterDsFile (Liste* file, Sommet* sommet) {
  sommet->marque = vrai;
  Succes* succes = new Succes();
  succes->somSuc = sommet;
  insererEnFinDeListe (file, succes);
}

// effectuer un parcours en largeur du graphe
void parcoursLargeur (Graphe* graphe) {
  printf ("\nParcours en largeur\n");
  razMarque (graphe);
  Liste* file = creerListe();
  for (int i=0; i<nbsommet (graphe); i++) {
    // somDepart : pointeur sur le sommet de départ
    Sommet* somDepart = getsommet (graphe, i);
    if (!somDepart->marque) {
      printf ("\n %s ", nomsommet (graphe, somDepart));
      ajouterDsFile (file, somDepart);

      while (!listeVide (file)) {
        Succes* succes = (Succes*) extraireEnTeteDeListe (file);
        somDepart = succes->somSuc;

        // remplacer dans la file le sommet de départ par ses successeurs
        Liste* li = &pSomD->li;
        ouvrirListe (li);
        while (!finListe (li) ) {
          succes = (Succes*) objetCourant (li);
          Sommet* somSuc = succes->somSuc;
          if (!somSuc->marque) {
            printf ("%s ", nomsommet (graphe, somSuc));
            ajouterDsFile (file, somSuc);
          }
        }  // while
      }  // while

    } // if
  } // for
}
```

5.5.8 Plus court chemin en partant d'un sommet

Il s'agit de trouver le plus court chemin pour aller d'un sommet vers les autres sommets. Cette méthode est connue sous le terme d'algorithme de Dijkstra (les coûts doivent être >= 0). Sur le graphe de la Figure 149, les plus courts chemins et leur coût en partant du sommet initial S0 sont les suivants :

Plus court chemin pour aller de S0 à :
S1 (cout = 25) : S0, S1
S2 (cout = 55) : S0, S1, S2
S3 (cout = 58) : S0, S1, S3
S5 (cout = 39) : S0, S6, S5
S6 (cout = 17) : S0, S6
De S0, on ne peut pas aller en S4, ni en S7.

Les fonctions *tousMarque()*, *dMin()* et *ecrireResultats()* sont des fonctions utilisées dans la fonction de calcul du plus court chemin. Les structures de données de la mémorisation du graphe sont celles des Figures 150 et 151.

```
// fournir vrai si tous les sommets du graphe G
// sont marqués, faux sinon
static booleen tousMarque (Graphe* graphe) {
  int i = 0;
  while ( i < nbsommet(graphe) && estmarque (graphe,i)) i++;
  return i >= nbsommet (graphe);
}
```

d est un tableau contenant le plus court chemin entre un sommet nsi (numéro du sommet initial) et chacun des autres sommets. d[0] est la valeur du plus court chemin de nsi à 0; d[1] la valeur du plus court de nsi à 1, etc. La fonction *dMin()* fournit le rang dans d de la plus petite valeur de d correspondant à un sommet non marqué.

```
// retourner l'indice de l'élément non marqué ayant le d[i] minimum
static int dMin (Graphe* graphe, int* d) {
  int min  = INFINI;
  int nMin = 0;
  for (int i=0; i<nbsommet(graphe); i++) {
    if (!estmarque (graphe,i)) {
      if (d[i] <= min) { min = d[i]; nMin = i; }
    }
  }
  return nMin;
}
```

La fonction *ecrireResultats()* écrit pour un sommet de départ nsi, le chemin le plus court entre nsi et les autres sommets. d[i] indique la valeur du chemin le plus court entre le sommet nsi et le sommet i (i != nsi). Si d[i] = INFINI (le plus grand entier noté * sur la Figure 152), c'est par convention qu'il n'y a pas de chemin de nsi à i. pr[i] indique quel est le sommet d'où on vient (avant-dernière étape) en prenant le chemin le plus court pour arriver à i.

Exemple :

	Sommet	d	pr
0	S0	*	2
1	S1	51	3
nsi = 2	S2	0	2
3	S3	18	2
4	S4	*	2
5	S5	66	1
6	S6	*	2
7	S7	*	2

Figure 152 Plus court chemin en partant de S2.

Pour aller du sommet S2 au sommet S3, le plus court chemin a pour valeur 18; pour aller de S2 à S5, le plus court chemin est 66. Pour aller de S2 à S4, il n'y a pas de chemin (valeur INFINI notée par une *). Pour aller de S2 à S2, le coût est 0.

Le tableau pr permet de reconstituer le chemin en partant du sommet d'arrivée.
Pour aller de S2 vers S5, le sommet précédant l'arrivée est le sommet 1 (pr[5]).
Pour aller de S2 vers S1, le sommet précédant l'arrivée est le sommet 3 (pr[1]).
Pour aller de S2 vers S3, le sommet précédant l'arrivée est le sommet 2 (pr[3]).
Le chemin de S2 à S5 est donc S2, S3, S1, S5.

La fonction *ecrireResultats()* écrit les valeurs des tableaux d et pr, et donne ensuite les chemins de l'arrivée vers le départ nsi pour tous les sommets différents de nsi et s'il existe un chemin (d[i] != INFINI).

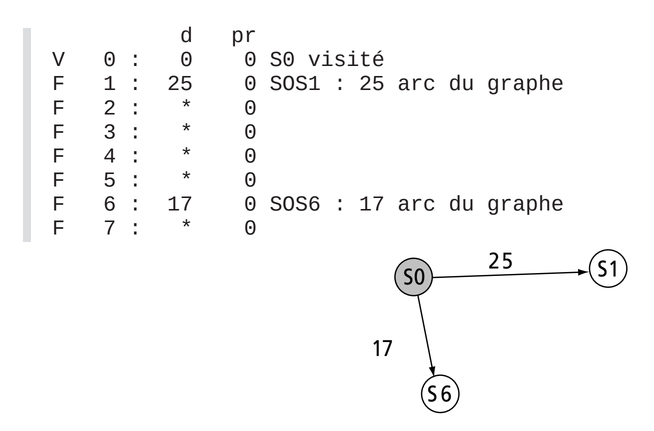
```
nsi S2
        S0 :   *  S2
        S1 :  51  S3
        S2 :   0  S2
        S3 :  18  S2
        S4 :   *  S2
        S5 :  66  S1
        S6 :   *  S2
        S7 :   *  S2

Plus court chemin (en partant de la fin) :
pour aller de S2 à :
  S1 (cout = 51) : S1, S3, S2
  S3 (cout = 18) : S3, S2
  S5 (cout = 66) : S5, S1, S3, S2
```

```
static void ecrireResultats (Graphe* graphe, int nsi, int* d, int* pr) {
  printf ("\nPlus court chemin (en partant de la fin) :\n");
  printf ("pour aller de %s à :\n", nomsommet(graphe,nsi));
  for (int i=0; i<nbsommet(graphe); i++) {
    if ( (i!=nsi) && (d[i] != INFINI) ) {
      printf ("  %s (cout = %d) : %s", nomsommet(graphe,i),
               d[i], nomsommet(graphe,i));
      int j = i;
      while (pr [j] != nsi) {
        printf (", %s", nomsommet (graphe, pr[j]));
        j = pr [j];
      }
      printf (", %s\n", nomsommet (graphe, pr[j]));
    }
  }
  printf ("\n");
}
```

La fonction *void* ***plusCourt*** *(Graphe* graphe, int nsi) ;* réalise le calcul du plus court chemin du graphe en partant du sommet nsi vers tous les autres sommets. La fonction utilise un tableau d (d[i] : plus court chemin entre nsi et i), et un tableau pr (qui indique le sommet précédemment visité pour arriver en i).

Si nsi vaut 0 (S0), au départ d et pr ont les valeurs suivantes sur l'exemple de la Figure 149 :



```
           d   pr
V   0 :    0    0 S0 visité
F   1 :   25    0 SOS1 : 25 arc du graphe
F   2 :    *    0
F   3 :    *    0
F   4 :    *    0
F   5 :    *    0
F   6 :   17    0 SOS6 : 17 arc du graphe
F   7 :    *    0
```


Figure 153 Recherche du plus court chemin : étape initiale.

On détermine le rang m du plus petit de d[i] non marqué soit : 6. On examine si en partant de S6, on peut trouver des chemins plus courts que ceux jusqu'à présent répertoriés. De S6, on peut aller en S5 (coût : 17+22=39 meilleur que ce que l'on connaît pour S5 qui est * donc inaccessible). Le sommet précédent pour arriver en S5 est donc 6 (S6).

```
nsi = 0, m = 6
           d   pr
V   0 :    0    0
F   1 :   25    0
F   2 :    *    0
F   3 :    *    0
F   4 :    *    0
F   5 :   39    6 pour arriver en 5, il faut passer par 6
V   6 :   17    0 S6 visité
F   7 :    *    0
```

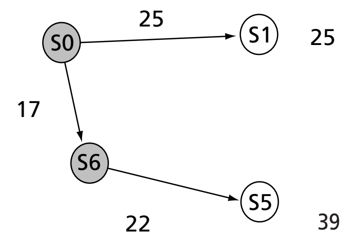


Figure 154 Plus court chemin : étape 1.

On détermine à nouveau le rang m du plus petit de d[i] non marqué soit : 1. On examine si en partant de S1, on peut trouver des chemins plus courts que ceux jusqu'à présent répertoriés. De S1, on peut aller en :

– S2 (coût : 25+30=55 meilleur que ce que l'on connaît pour S2 qui est * donc inaccessible) ; le sommet précédent pour arriver en S2 est donc 1 (S1),
– S3 (coût : 25+33=58 meilleur que ce que l'on connaît pour S3 qui est * donc inaccessible) ; le sommet précédent pour arriver en S3 est donc 1 (S1),
– S5 (coût : 25+15=40 supérieur au coût déjà connu 39, donc non retenu).

```
nsi = 0, m = 1
           d   pr
V   0 :   0    0
V   1 :  25    0 S1 visité
F   2 :  55    1 pour arriver en 2, il faut passer par 1
F   3 :  58    1 pour arriver en 3, il faut passer par 1
F   4 :   *    0
F   5 :  39    6
V   6 :  17    0
F   7 :   *    0
```

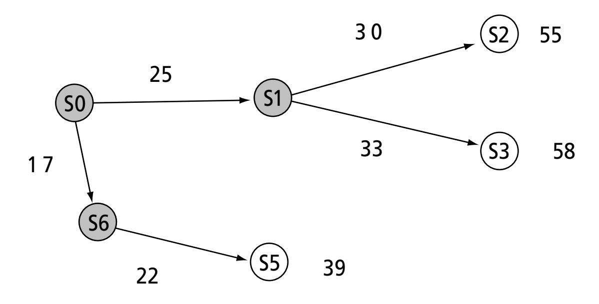


Figure 155 Plus court chemin : étape 2.

On détermine à nouveau le rang du plus petit de d[i] non marqué soit : 5. De S5, on peut aller vers S1 (déjà marqué) ou vers S3 par un chemin plus long. Le sommet 5 est marqué.

On détermine à nouveau le rang du plus petit de d[i] non marqué soit : 2. De S2, on peut aller vers S3 par un chemin plus long. Le sommet 2 est marqué.

On détermine à nouveau le rang du plus petit de d[i] non marqué soit : 3. De S3, on peut aller vers S1 déjà marqué. Le sommet 3 est marqué.

Les sommets 4 et 7 sont inaccessibles (noté *) en partant de S0.

```
// plus court chemin en partant du sommet nsi
void plusCourt (Graphe* graphe, int nsi) {
  // allocation dynamique des tableaux d et pr
  int* d  = (int*) malloc (sizeof (int) * nbsommet(graphe));
  int* pr = (int*) malloc (sizeof (int) * nbsommet(graphe));

  // initialisation par défaut de d et pr
  razMarque (graphe);
  for (int i=0; i<nbsommet(graphe); i++) {
    d  [i] = INFINI;
    pr [i] = nsi;
  }
  d [nsi] = 0;
```

```
  // initialisation de d et pr en fonction de graphe
  Liste* li = &(getsommet(graphe,nsi)->li);
  ouvrirListe (li);
  while (!finListe (li) ) {
    Succes* succes = (Succes*) objetCourant (li);
    int i = succes->somSuc->num;
    //printf ("num %d\n", i);
    d [i] = succes->cout;
  }


  printf ("NSI : %d\n", nsi);
  marquersommet (graphe,nsi); // marquer NSI

  while (!tousMarque (graphe)) {
    int m = dMin (graphe, d); // élément minimum non marqué de d
    marquersommet (graphe, m);

    if (d [m] != INFINI) {
      li = &getsommet(graphe,m)->li;
      ouvrirListe (li);
      while (!finListe (li) ) {
        Succes* succes = (Succes*) objetCourant (li);
        int k = succes->somSuc->num;
        if (!estmarque (graphe,k)) {
          int v = d [m] + succes->cout;
          if (v < d [k]) {
            d [k]  = v;
            pr [k] = m;
          }
        }
      }
    }
  }
  ecrireResultats (graphe, nsi, d, pr);
}
```

5.5.9 Création d'un graphe à partir d'un fichier

L'initialisation d'un graphe peut se faire à partir d'une description contenue dans un fichier. *lireGraphe()* initialise le graphe, lit les noms des sommets et des relations entre ces sommets pour construire le graphe. La fonction utilise *ajouterUnSommet()* et *ajouterUnArc()* vues précédemment. *lireUnMot()* effectue la lecture d'un nom de sommet en ignorant les espaces avant et après le nom.

Ce programme est le même que pour le test des graphes mémorisés sous forme de matrices (ci-après). Les seules différences sont repérées par la variable de compilation MATRICE.

```
/* liregraphe.cpp  créer un graphe à partir d'une description
   du graphe faite dans un fichier
   pour liste d'adjacence ou matrices
   suivant la variable de compilation MATRICE */

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
```

```
#ifndef MATRICE
#include "grapheadj.h"
#else
#include "graphemat.h"
typedef GrapheMat Graphe;
#endif

typedef char NomSom [20];
int c;   // un caractère lu en avance dans lireUnMot

// ignorer les blancs
void lireBlancs (FILE* fe) {
  while ( ( (c==' ')  || (c=='\n') || (c==13) )  && !feof(fe) ) {
    c = getc(fe);
  }
}

// lire un nom de sommet en ignorant les espaces
void lireUnMot (FILE* fe, char* chaine) {
  char* pCh = chaine;

  //printf ("Debut lireUnMot %c %d\n", c, c);
  lireBlancs (fe); // blancs avant le mot
  while ( isalpha(c) || isdigit(c) ) {
    *pCh++ = (char) c;
    //printf ("-- %c %d\n", c, c);
    c = getc(fe);
  }
  *pCh = 0;
  lireBlancs (fe);  // blancs après le mot
  //printf ("Fin lireUnMot %s\n", chaine);  getchar();
}
```

Si les sommets sont suivis de valeurs entre parenthèses, le graphe est déclaré valué sinon le graphe n'est pas valué (voir les fichiers correspondant au graphe non valué et valué de la Figure 149).

Graphe non valué

```
S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 S6 ;
S1 : S2 S3 S5 ;
S2 : S3 ;
S3 : S1 ;
S4 : S3 S5 S7 ;
S5 : S1 S3 ;
S6 : S5 ;
```

Graphe valué

```
S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 (25)  S6 (17) ;
S1 : S2 (30)  S3 (33)  S5 (15) ;
S2 : S3 (18) ;
S3 : S1 (33) ;
S4 : S3 (25)  S5 (26)  S7 (20) ;
S5 : S1 (15)  S3 (35) ;
S6 : S5 (22) ;
```

```
// fournir un pointeur sur un graphe construit
// à partir d'un fichier fe de données
// value = vrai si le Graphe est valué
Graphe* lireGraphe (FILE* fe, int nMaxSom) {
  booleen value = faux;
  #ifndef MATRICE
  Graphe* graphe = creerGraphe (nMaxSom, faux);
  #else
  Graphe* graphe = creerGrapheMat (nMaxSom, faux);
  #endif
```

```
    // lire les noms des sommets
    c = getc(fe);  // c global
    while (c != ';') {
      char* somD = (char*) malloc (20);
      lireUnMot (fe, somD);
      ajouterUnSommet (graphe, somD);
    }

    while (c != EOF) {
      c = getc(fe); // passe ;
      NomSom somD;

      lireUnMot (fe, somD); // lit le sommet de départ
      if (c != ':') {
        if (c != EOF) printf ("Manque : %c (%d)\n", c,c);
        graphe->value = value;
        return graphe;
      }

      c = getc(fe);
      while (c != ';') {
        NomSom somA;
        lireUnMot (fe, somA); // lit les sommets d'arrivée
        int cout;
        if (c == '(') {
          value = vrai; // si sommet suivi de ( : S1(25)
          fscanf (fe, "%d", &cout);
          c = getc (fe); // passer )
          if (c != ')') printf ("Manque )\n");
          c = getc (fe);
          lireBlancs (fe); // prochain à analyser
          //printf ("cout %d\n", cout);
        } else {
          cout = 0;
        }
        ajouterUnArc (graphe, somD, somA, cout);
      }
    }
    graphe->value = value;
    return graphe;
}
```

5.5.10 Menu de test des graphes (listes d'adjacence et matrices)

Le menu suivant permet de créer un graphe, d'ajouter des sommets ou des arcs (relations) à ce graphe, d'afficher les parcours en profondeur ou en largeur, et de calculer les plus courts chemins en partant des différents sommets du graphe.

```
/* ppgraphe.cpp pour listes et matrices
                MATRICE permet de compiler l'un ou l'autre */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef MATRICE
#include "grapheadj.h"
typedef char NomSom [20];  // défini dans graphemat.h
#else
#include "graphemat.h"
typedef GrapheMat Graphe;
#endif
```

```
int menu () {
  #ifndef MATRICE
  printf ("\n\nGRAPHES avec des listes d'adjacence\n\n");
  #else
  printf ("\n\nGRAPHES avec matrices\n\n");
  #endif
  printf ("0 - Fin du programme\n");
  printf ("1 - Création à partir d'un fichier\n");
  printf ("\n");
  printf ("2 - Initialisation d'un graphe vide\n");
  printf ("3 - Ajout d'un sommet\n");
  printf ("4 - Ajout d'un arc\n");
  printf ("\n");
  printf ("5 - Liste des sommets et des arcs\n");
  printf ("6 - Destruction du graphe\n");
  printf ("7 - Parcours en profondeur d'un graphe\n");
  printf ("8 - Parcours en largeur    d'un graphe\n");
  printf ("\n");
  #ifndef MATRICE
    printf ("9 - Les plus courts chemins\n");
  #else
    printf ("9  - Floyd\n");
    printf ("10 - Produit et fermeture\n");
    printf ("11 - Warshall\n");
  #endif
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  printf ("\n");
  return cod;
}

void main () {
  Graphe* graphe;
  booleen fini = faux;

  while (!fini) {

    switch ( menu() ) {

    case 0:
      fini = vrai;
      break;

    case 1: {  // création à partir d'un fichier
      printf ("Nom du fichier contenant le graphe ? ");
      char nomFe [50];
      //scanf  ("%s", nomFe);
      strcpy (nomFe, "graphe1.dat");
      FILE* fe = fopen (nomFe, "r");
      if (fe == NULL) {
        perror (nomFe);
      } else {
        graphe = lireGraphe (fe, 20); // 20 sommets maximum
        fclose (fe);
      }
      } break;
```

```
case 2: {             // création d'un graphe vide
  printf ("Nombre maximum de sommets ? ");
  int nMaxSom; scanf  ("%d", &nMaxSom);
  printf ("0) graphe valué; 1) non valué ? ");
  int value; scanf ("%d", &value);
  #ifndef MATRICE
  graphe = creerGraphe (nMaxSom, value);
  #else
  graphe = creerGrapheMat (nMaxSom, value);
  #endif
} break;

case 3: {  // ajouter un sommet
  printf ("Nom du sommet à insérer ? ");
  NomSom somD; scanf  ("%s", somD);
  ajouterUnSommet (graphe, somD);
} break;

case 4: {  // ajouter un arc
  printf ("Nom du sommet de départ ? ");
  NomSom somD; scanf  ("%s", somD);
  printf ("Nom du sommet d'arrivée ? ");
  NomSom somA; scanf  ("%s", somA);
  int cout;
  if (graphe->value) {
    printf ("Cout de la relation ? ");
    scanf ("%d", &cout);
  } else {
    cout = 0;
  }
  ajouterUnArc (graphe, somD, somA, cout);
} break;

case 5:
  ecrireGraphe (graphe);
  break;

case 6:
  detruireGraphe (graphe);
  break;

case 7:
  parcoursProfond (graphe);
  break;

case 8:
  parcoursLargeur (graphe);
  break;

#ifndef MATRICE
case 9:
  if (graphe->value) {
    printf ("\nLes plus courts chemins\n\n");
    for (int i=0; i<graphe->table->n; i++) {
      plusCourt (graphe, i);  getchar();
    }
  } else {
    printf ("Graphe non valué\n");
  }
  break;
#else
```

```
      case 9:
        if (graphe->value) {
          printf ("\nLes plus courts chemins\n\n");
          floyd (graphe);
        } else {
          printf ("Graphe non valué\n");
        }
        break;

      case 10:
        produitEtFermeture (graphe);
        break;

      case 11:
        warshall (graphe);
        break;

      #endif

      }   // switch

      if (!fini) {
        printf ("\n\nTaper Return pour continuer\n");
        getchar();
      }
    }
  }
```

Exemple d'interrogation concernant les graphes :

```
GRAPHES avec des listes d'adjacence

0 - Fin du programme
1 - Création à partir d'un fichier

2 - Initialisation d'un graphe vide
3 - Ajout d'un sommet
4 - Ajout d'un arc

5 - Liste des sommets et des arcs
6 - Destruction du graphe
7 - Parcours en profondeur d'un graphe
8 - Parcours en largeur    d'un graphe

9 – Les plus courts chemins

Votre choix ? 5

graphe valué

S0 S1 S2 S3 S4 S5 S6 S7 ;
S0 : S1 ( 25) S6 ( 17) ;
```

```
S1 : S2 ( 30) S3 ( 33) S5 ( 15) ;
S2 : S3 ( 18) ;
S3 : S1 ( 33) ;
S4 : S3 ( 25) S5 ( 26) S7 ( 20) ;
S5 : S1 ( 15) S3 ( 35) ;
S6 : S5 ( 22) ;
S7 : ;
```

Pour obtenir les plus courts chemins de tous les sommets vers tous les autres sommets, il suffit d'appeler la fonction de calcul du plus court chemin avec pour sommet initial S0, puis S1, etc. Les résultats sont donnés ci-dessous en chemin inverse (du sommet d'arrivée vers le sommet de départ).

```
pour aller de S0 à :
  S1 (cout = 25) : S1, S0
  S2 (cout = 55) : S2, S1, S0
  S3 (cout = 58) : S3, S1, S0
  S5 (cout = 39) : S5, S6, S0
  S6 (cout = 17) : S6, S0

pour aller de S1 à :
  S2 (cout = 30) : S2, S1
  S3 (cout = 33) : S3, S1
  S5 (cout = 15) : S5, S1

pour aller de S2 à :
  S1 (cout = 51) : S1, S3, S2
  S3 (cout = 18) : S3, S2
  S5 (cout = 66) : S5, S1, S3, S2

pour aller de S3 à :
  S1 (cout = 33) : S1, S3
  S2 (cout = 63) : S2, S1, S3
  S5 (cout = 48) : S5, S1, S3

pour aller de S4 à :
  S1 (cout = 41) : S1, S5, S4
  S2 (cout = 71) : S2, S1, S5, S4
  S3 (cout = 25) : S3, S4
  S5 (cout = 26) : S5, S4
  S7 (cout = 20) : S7, S4
```

```
pour aller de S5 à :
  S1 (cout = 15) : S1, S5
  S2 (cout = 45) : S2, S1, S5
  S3 (cout = 35) : S3, S5

pour aller de S6 à :
  S1 (cout = 37) : S1, S5, S6
  S2 (cout = 67) : S2, S1, S5, S6
  S3 (cout = 57) : S3, S5, S6
  S5 (cout = 22) : S5, S6
```

5.6 MÉMORISATION SOUS FORME DE MATRICES

Voir § 5.3.1, page 259 : mémorisation sous forme de matrices d'adjacence.

5.6.1 Le fichier d'en-tête du module des graphes (matrices)

Une structure de type **GrapheMat** mémorisée sous forme d'une matrice contient :

- les variables n (nombre réel de sommets), nMax (nombre maximum de sommets) et value (vrai si le graphe est valué),
- les tableaux nécessaires à la mémorisation du graphe : un tableau nomS des noms des sommets, un tableau *element* de booléens indiquant les relations entre les sommets, un tableau *valeur* indiquant le coût des relations, un tableau *marque* pour le parcours du graphe, voir Figure 146, page 260.

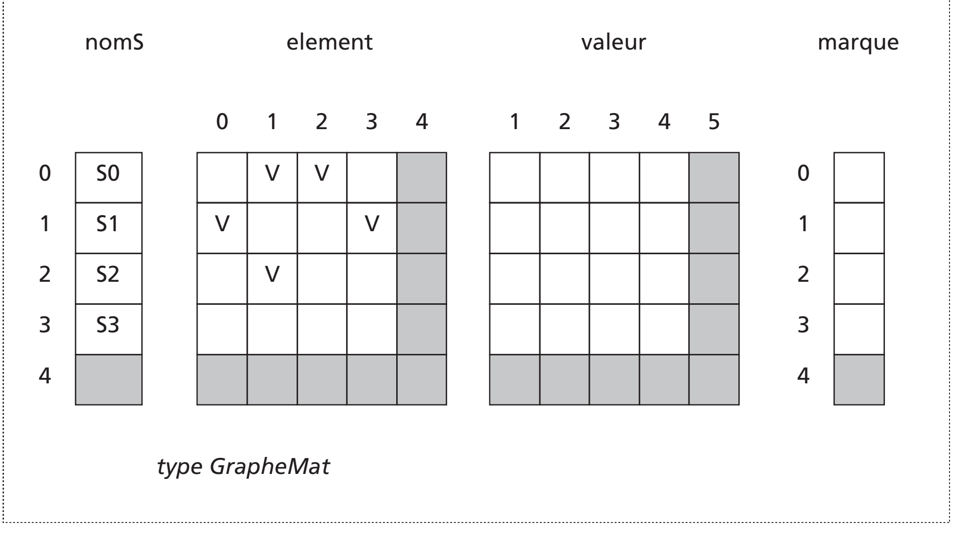


Figure 156 Le type GrapheMat.

Les prototypes des fonctions sont les mêmes que pour le type Graphe mémorisé sous forme de listes d'adjacence. Cependant la création d'un graphe se fait pour une matrice avec *creerGrapheMat()* ; la fonction *PlusCourt()* est remplacée par la fonction *Floyd()* et quelques nouvelles fonctions sont ajoutées.

```
/* graphemat.h
   pour la gestion de graphes mémorisés sous forme de matrices */

#ifndef GRAPHEMAT_H
#define GRAPHEMAT_H

#include <stdio.h>

typedef int  booleen;
#define faux 0
#define vrai 1
typedef char NomSom[20];  // nom d'un sommet
#define INFINI INT_MAX

typedef int* Matrice;

typedef struct {
  int      n;       // nombre de sommets
  int      nMax;    // nombre max de sommets
  booleen  value;   // graphe valué ou non
  NomSom*  nomS;    // noms des sommets
  Matrice  element; // existence d'un arc (i, j)
  Matrice  valeur;  // cout de l'arc (i, j)
  booleen* marque;  // sommet marqué (visité) ou non
} GrapheMat;

GrapheMat* creerGrapheMat     (int nMax, int value);
void       detruireGraphe     (GrapheMat* graphe);
void       ajouterUnSommet    (GrapheMat* graphe, NomSom nom);
void       ajouterUnArc       (GrapheMat* graphe, NomSom somD,
                                                  NomSom somA, int cout);
GrapheMat* lireGraphe         (FILE*   fe, int nMaxSom);
void       ecrireGraphe       (GrapheMat* graphe);

void       parcoursProfond    (GrapheMat* graphe);
void       parcoursLargeur    (GrapheMat* graphe);
void       floyd              (GrapheMat* graphe);
void       produitEtFermeture (GrapheMat* graphe);
void       warshall           (GrapheMat* graphe);

#endif
```

5.6.2 Création et destruction d'un graphe (matrices)

La fonction *creerGrapheMat()* alloue dynamiquement et initialise une structure de type GrapheMat. Si une relation n'existe pas entre 2 sommets i et j, la valeur est notée INFINI (plus grand entier sur ordinateur). La fonction *detruireGraphe()* désalloue une structure de type GrapheMat allouée avec *creerGrapheMat()*.

```
// remise à zéro du tableau de marquage
static void razMarque (GrapheMat* graphe) {
```

```
  for (int i=0; i<graphe->n; i++) graphe->marque [i] = faux;
}

// création d'une variable de type GrapheMat;
// nMax  : nombre maximum de sommets envisagés;
// value : vrai si le graphe est valué
GrapheMat* creerGrapheMat (int nMax, int value) {

  // allocation de graphe
  GrapheMat* graphe  = (GrapheMat*) malloc (sizeof (GrapheMat));
  graphe->n       = 0;
  graphe->nMax    = nMax;
  graphe->value   = value;
  graphe->nomS    = (NomSom*)  malloc (sizeof(NomSom)  *nMax);
  graphe->marque  = (booleen*) malloc (sizeof(booleen) *nMax);
  graphe->element = (int*)     malloc (sizeof(int)*nMax*nMax);
  graphe->valeur  = (int*)     malloc (sizeof(int)*nMax*nMax);

  // initialisation par défaut
  for (int i=0; i<nMax; i++) {
    for (int j=0; j<nMax; j++) {
      graphe->element [i*nMax+j] = faux;
      graphe->valeur  [i*nMax+j] = INFINI;
    }
  }
  for (int i=0; i<nMax; i++) graphe->valeur [i*nMax+i] = 0;
  razMarque (graphe);

  return graphe;
}

// désallocation d'un graphe
void detruireGraphe (GrapheMat* graphe) {
  free (graphe->nomS);
  free (graphe->marque);
  free (graphe->element);
  free (graphe->valeur);
  free (graphe);
}
```

5.6.3 Insertion d’un sommet ou d’un arc dans un graphe (matrices)

La fonction *rang()* fournit le rang dans le tableau nomS du nom.

```
static int rang (GrapheMat* graphe, NomSom nom) {
  int      i      = 0;
  booleen  trouve = faux;

  while ( (i<graphe->n) && !trouve) {
    trouve = strcmp (graphe->nomS [i], nom) == 0;
    if (!trouve) i++;
  }
  return trouve ? i : -1;
}
```

La fonction *ajouterUnSommet()* ajoute le sommet nom au graphe.

```
void ajouterUnSommet (GrapheMat* graphe, NomSom nom) {
  if (rang (graphe, nom) == -1) {
```

```
    if (graphe->n < graphe->nMax) {
     strcpy (graphe->nomS [graphe->n++], nom);
    } else {
      printf ("\nNombre de sommets > %d\n", graphe->nMax);
    }
  } else {
    printf ("\n%s déjà défini\n", nom);
  }
}
```

La fonction *ajouterUnArc()* ajoute au graphe un arc entre somD et somA.

```
void ajouterUnArc (GrapheMat* graphe, NomSom somD, NomSom somA, int cout) {
  int nMax = graphe->nMax;
  int rd = rang (graphe, somD);
  int rg = rang (graphe, somA);
  graphe->element [rd*nMax+rg] = vrai;
  graphe->valeur  [rd*nMax+rg] = cout;
}
```

5.6.4 Lecture d'un graphe (à partir d'un fichier)

La fonction *lireGraphe()* pour la mémorisation sous forme de matrices est identique à *lireGraphe()* (voir § 5.5.9, page 274) déjà vue pour les listes d'adjacence, à quelques exceptions près (voir ifdef MATRICE). *lireGraphe()* utilise les fonctions *ajouterUnSommet()*, *ajouterUnArc()* qui sont redéfinies (avec les mêmes prototypes) pour la mémorisation en matrices. La création du graphe se fait avec *creerGraphe()* ou *creerGrapheMat()*.

5.6.5 Écriture d'un graphe

L'écriture d'un graphe mémorisé sous forme de matrice :

```
void ecrireGraphe (GrapheMat* graphe) {
  int nMax = graphe->nMax;

  for (int i=0; i<graphe->n; i++) printf ("%s ", graphe->nomS[i]);
  printf (";\n");

  for (int i=0; i<graphe->n; i++) {
    printf ("\n%s : ", graphe->nomS[i]);
    for (int j=0; j<graphe->n; j++) {
      if (graphe->element [i*nMax+j] == vrai) {
        printf ("%s ", graphe->nomS[j]) ;
        if (graphe->value) {
          printf (" (%3d) ", graphe->valeur [i*nMax+j] );
        }
      }
    }
    printf (";");
  }
}
```

5.6.6 Parcours en profondeur (matrices)

La fonction de parcours en profondeur se réécrit très facilement avec cette nouvelle structure de données (voir § 5.4.1, page 260 et 5.5.6, page 268).

```
static void profondeur (GrapheMat* graphe, int numSommet, int niveau) {
  int nMax = graphe->nMax;
  graphe->marque [numSommet] = vrai;
  for (int i=1; i<niveau; i++) printf ("%5s", " ");
  printf ("%s\n", graphe->nomS [numSommet]);

  for (int i=0; i<graphe->n; i++) {
    if ( (graphe->element [numSommet*nMax+i] == vrai)
          && !graphe->marque [i] ) {
      profondeur (graphe, i, niveau+1);
    }
  }
}

// marque pourrait contenir le numéro d'ordre de visite du sommet;
// -1 si pas visité; numéro d'ordre sinon
void parcoursProfond (GrapheMat* graphe) {
  razMarque (graphe);
  for (int i=0; i<graphe->n; i++) {
    if (!graphe->marque [i]) profondeur (graphe, i, 1);
  }
}
```

5.6.7 Parcours en largeur (matrices)

Le parcours en largeur nécessite une liste (file d'attente) des éléments à traiter (voir § 5.4.2, page 262).

```
// ajouter le numéro numS du sommet dans la file d'attente file
static void ajouterDsFile (GrapheMat* graphe, Liste* file, int numS) {
  graphe->marque [numS] = vrai;
  Succes* nouveau = (Succes*) malloc (sizeof (Succes) );
  nouveau->numSom = numS;
  insererEnFinDeListe (file, nouveau);
}

// effectuer un parcours en largeur du graphe
void parcoursLargeur (GrapheMat* graphe) {
  int nMax = graphe->nMax;

  razMarque (graphe);
  Liste*  file = creerListe();

  for (int i=0; i<graphe->n; i++) {
    if (!graphe->marque[i]) {
      printf ("\n %s ", graphe->nomS [i]);
      ajouterDsFile (graphe, file, i);

      while (!listeVide(file)) {
        Succes* succes = (Succes*) extraireEnTeteDeListe (file);
        int s = succes->numSom;
```

```
          // insérer dans file, les successeurs de s
          for (int j=0; j<graphe->n; j++) {
           if ( (graphe->element [s*nMax+j] == vrai) && !graphe->marque[j] ) {
              printf (" %s ", graphe->nomS [j]);
              ajouterDsFile (graphe, file, j);
            }
          }
        } // while
      }
    } // for
  }
```

5.6.8 Plus courts chemins entre tous les sommets (Floyd)

Soit le graphe valué de la Figure 149, page 261. À partir du type GrapheMat (matrice), on peut créer les 2 matrices a et p suivantes indiquant le coût et le dernier sommet visité. Ainsi a(0,1) = 25 indique le coût de la relation entre S0 et S1 et p(0,1) = 0 indique le numéro du sommet 0 d'où on vient quand on va de S0 à S1. a(0,2) = INFINI (noté *) car il n'y a pas de relation directe entre S0 et S2 ; p(0,2) est mis à 0 mais il n'est pas significatif dans ce cas.

```
a : matrice initiale de coût          p : dernier sommet visité
   0  25   *   *   *   *  17   *         0  0  0  0  0  0  0  0
   *   0  30  33   *  15   *   *         1  1  1  1  1  1  1  1
   *   *   0  18   *   *   *   *         2  2  2  2  2  2  2  2
   *  33   *   0   *   *   *   *         3  3  3  3  3  3  3  3
   *   *   *  25   0  26   *  20         4  4  4  4  4  4  4  4
   *  15   *  35   *   0   *   *         5  5  5  5  5  5  5  5
   *   *   *   *   *  22   0   *         6  6  6  6  6  6  6  6
   *   *   *   *   *   *   *   0         7  7  7  7  7  7  7  7
```

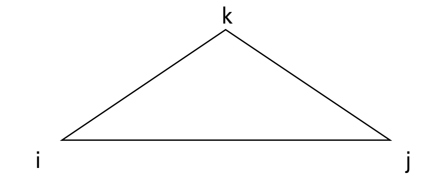


Figure 157 Principe de l'algorithme de Floyd.

L'algorithme de Floyd envisage pour chaque arc (i, j) si un passage par un sommet k peut raccourcir la distance entre i et j dans le cas où les chemins (i, k) et (k, j) existent, ces chemins n'étant pas forcément élémentaires, mais pouvant se constituer d'un chemin contenant des numéros de sommets inférieurs à k.

Voici ci-dessous les différentes étapes de l'exemple :

```
Passage par le sommet numéro 0
```

Comme on ne peut accéder à S0 en partant d'un autre sommet sur l'exemple, la tentative de trouver des plus courts chemins en passant par S0 échoue.

```
Passage par le sommet numéro 1
a : matrice de coût                      p : dernier sommet visité
   0  25  55  58   *  40  17   *            0  0  1  1  0  1  0  0
   *   0  30  33   *  15   *   *            1  1  1  1  1  1  1  1
   *   *   0  18   *   *   *   *            2  2  2  2  2  2  2  2
   *  33  63   0   *  48   *   *            3  3  1  3  3  1  3  3
   *   *   *  25   0  26   *  20            4  4  4  4  4  4  4  4
   *  15  45  35   *   0   *   *            5  5  1  5  5  5  5  5
   *   *   *   *   *  22   0   *            6  6  6  6  6  6  6  6
   *   *   *   *   *   *   *   0            7  7  7  7  7  7  7  7
```

Le passage par le sommet 1 améliore les relations suivantes qui étaient INFINI.

S0-S2 S0-S1-S2 : 55 p[0][2] = p[1][2] = 1 avant dernière étape de S1-S2
S0-S3 S0-S1-S3 : 58 p[0][3] = p[1][3] = 1 avant dernière étape de S1-S3
S0-S5 S0-S1-S5 : 40 p[0][5] = p[1][5] = 1 avant dernière étape de S1-S5
S3-S2 S3-S1-S2 : 63 p[3][2] = p[1][2] = 1 avant dernière étape de S1-S2
S3-S5 S3-S1-S5 : 48 p[3][5] = p[1][5] = 1 avant dernière étape de S1-S5
S5-S2 S5-S1-S2 : 45 p[5][2] = p[1][2] = 1 avant dernière étape de S1-S2

```
Passage par le sommet numéro 2
```

La seule relation aboutissant en S2 est une relation directe S1-S2. Le passage par 2 n'apporte pas de nouvelles solutions.

```
Passage par le sommet numéro 3
a : matrice de coût                      p : dernier sommet visité
   0  25  55  58   *  40  17   *            0  0  1  1  0  1  0  0
   *   0  30  33   *  15   *   *            1  1  1  1  1  1  1  1
   *  51   0  18   *  66   *   *            2  3  2  2  2  1  2  2
   *  33  63   0   *  48   *   *            3  3  1  3  3  1  3  3
   *  58  88  25   0  26   *  20            4  3  1  4  4  4  4  4
   *  15  45  35   *   0   *   *            5  5  1  5  5  5  5  5
   *   *   *   *   *  22   0   *            6  6  6  6  6  6  6  6
   *   *   *   *   *   *   *   0            7  7  7  7  7  7  7  7
```

Le passage par le sommet 3 améliore les relations suivantes qui étaient INFINI :

S2-S1 S2-S3-S1 : 51 p[2][1] = p[3][1] = 3 avant dernière étape de S3-S1
S2-S5 S2-S3-S5 : 66 p[2][5] = p[3][5] = 1 avant dernière étape de S3-S5
S4-S1 S4-S3-S1 : 58 p[4][1] = p[3][1] = 3 avant dernière étape de S3-S1
S4-S2 S4-S3-S2 : 88 p[4][2] = p[3][2] = 1 avant dernière étape de S3-S2

L'étape précédente du chemin S2-S3-S5 est l'étape précédente du chemin S3-S5 soit S1 car p[3][5] vaut 1. De même, le chemin S4-S2 est amélioré en passant par S3 : S4-S3-S2; cependant S3-S2 se fait en passant par S1, donc l'avant-dernière étape est 1 pour S4-S2.

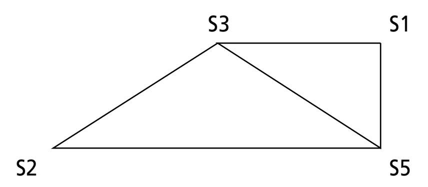


Figure 158 Chemin de S2 à S5.

```
Passage par le sommet numéro 4 : pas de nouveaux chemins

Passage par le sommet numéro 5
a : matrice de coût                    p : dernier sommet visité
   0  25  55  58   *  40  17   *          0  0  1  1  0  1  0  0
   *   0  30  33   *  15   *   *          1  1  1  1  1  1  1  1
   *  51   0  18   *  66   *   *          2  3  2  2  2  1  2  2
   *  33  63   0   *  48   *   *          3  3  1  3  3  1  3  3
   *  41  71  25   0  26   *  20          4  5  1  4  4  4  4  4
   *  15  45  35   *   0   *   *          5  5  1  5  5  5  5  5
   *  37  67  57   *  22   0   *          6  5  1  5  6  6  6  6
   *   *   *   *   *   *   *   0          7  7  7  7  7  7  7  7

Passage par le sommet numéro 6
a : matrice de coût                    p : dernier sommet visité
   0  25  55  58   *  39  17   *          0  0  1  1  0  6  0  0
   *   0  30  33   *  15   *   *          1  1  1  1  1  1  1  1
   *  51   0  18   *  66   *   *          2  3  2  2  2  1  2  2
   *  33  63   0   *  48   *   *          3  3  1  3  3  1  3  3
   *  41  71  25   0  26   *  20          4  5  1  4  4  4  4  4
   *  15  45  35   *   0   *   *          5  5  1  5  5  5  5  5
   *  37  67  57   *  22   0   *          6  5  1  5  6  6  6  6
   *   *   *   *   *   *   *   0          7  7  7  7  7  7  7  7

Passage par le sommet numéro 7 : pas de nouveaux chemins.
```

Les chemins inverses obtenus sont les mêmes que ceux du § 5.5.10, page 280, la disposition étant la même. Exemples de reconstitution de chemins :

Chemin de S2 à S5 : p (2, 5) = 1; p (2, 1) = 3; p (2, 3) = 2

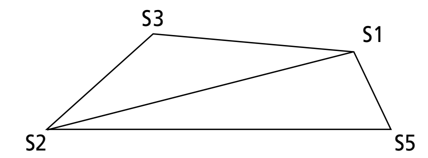


Figure 159 Reconstitution du plus court chemin de S2 à S5.

5.6.9 Algorithme de Floyd

On peut décomposer cet algorithme en différentes fonctions. *ecrireEtape()* permet d'écrire les deux matrices a et p après chaque tentative de passage systématique par le sommet k comme vu précédemment. Si k=-1, il s'agit de l'étape d'initialisation.

```
static void ecrireEtape (Matrice a, Matrice p, int k, int ns, int nMax) {
  if (k==-1) {
    printf ("Matrices initiales de cout et de dernier sommet visité\n");
  } else {
    printf ("Passage par le sommet numéro %d\n", k);
  }
  for (int i=0; i<ns; i++) {
    for (int j=0; j<ns; j++) {
      if (a [i*nMax+j]==INFINI) {
        printf (" %3s", "*");
```

```
      } else {
        printf (" %3d", a [i*nMax+j]);
      }
    }
    printf ("%6s", " ");
    for (int j=0; j<ns; j++) {
      printf ("%3d", p [i*nMax+j]);
    }
    printf ("\n");
  }
  printf ("\n");
}

// écrire les plus courts chemins en consultant les tableaux a et p
static void ecrirePlusCourt (GrapheMat* graphe, Matrice a, Matrice p) {
  int nMax = graphe->nMax;

  printf ("\n\nPlus court chemin (Floyd)\n");
  for (int i=0; i<graphe->n; i++) {
    printf ("pour aller de %s à :\n", graphe->nomS[i] );
    for (int j=0; j<graphe->n; j++) {
      if ((i != j) && (a [i*nMax+j] != INFINI) ) {
        printf ("  %s  (cout = %d) : ",
                graphe->nomS[j], a [i*nMax+j]);
        int k = p [i*nMax+j];
        printf ("%s, %s", graphe->nomS[j], graphe->nomS[k]);
        while (k != i) {
          k = p [i*nMax+k];
          printf (", %s ", graphe->nomS[k]);
        }
        printf ("\n");
      }
    }
    printf ("\n");
  }
  printf ("\n");
}

// initialiser les matrices a et p à partir de graphe
static void initFloyd (GrapheMat* graphe, Matrice* a, Matrice* p, int* ns) {
  int nMax = graphe->nMax;

  Matrice ta = (int*) malloc (sizeof(int)*nMax*nMax);
  Matrice tp = (int*) malloc (sizeof(int)*nMax*nMax);

  *ns = graphe->n;
  for (int i=0; i<graphe->n; i++) {
    for (int j=0; j<graphe->n; j++) {
      ta [i*nMax+j] = graphe->valeur [i*nMax+j];
      tp [i*nMax+j] = i;
    }
  }
  *a = ta;
  *p = tp;
}

// Cœur de l'algorithme : calcul des plus courts chemins
// pour chaque élément de la matrice; on effectue une tentative
// de passage par le kième sommet d'où 3 boucles imbriquées
void floyd (GrapheMat* graphe) {
  Matrice a, p;
  int     ns;
```

```
  int     nMax = graphe->nMax;

  initFloyd (graphe, &a, &p, &ns);
  ecrireEtape (a, p, -1, ns, graphe->nMax);

  for (int k=0; k<ns; k++) {
    for (int i=0; i<ns; i++) {
      for (int j=0; j<ns; j++) {
        if ( (a [i*nMax+k] != INFINI) &&
             (a [k*nMax+j] != INFINI) &&
             (a [i*nMax+k] + a [k*nMax+j] < a [i*nMax+j]) ) {
          a [i*nMax+j] = a [i*nMax+k] + a [k*nMax+j];
          p [i*nMax+j] = p [k*nMax+j];
        }
      }
    }
    ecrireEtape (a, p, k, ns, graphe->nMax);
  }

  ecrirePlusCourt (graphe, a, p);
  free (a);
  free (p);
}
```

5.6.10 Algorithme de calcul de la fermeture transitive

La fermeture transitive permet de connaître *l'existence* d'un chemin de longueur quelconque entre 2 sommets i et j.

Si M est la matrice représentant l'existence d'un chemin élémentaire entre i et j, le produit :

- M^2 = M*M représente l'existence d'un chemin de longueur 2 entre i et j ;
- M^3 = M*M*M, l'existence d'un chemin de longueur 3.
- S'il y a N sommets, on peut calculer jusqu'à M^N (M à la puissance N).

La somme SM des matrices M + M2 + ... + MN représente l'existence d'un chemin de longueur 1, 2, ... ou N, entre i et j. C'est la fermeture transitive ainsi appelée car il y a transitivité dans l'existence de chemins : s'il existe un chemin de i à j, et s'il existe un chemin de j à k, il existe un chemin de i à k.

Exemple du produit booléen :

M^2 (1, 1) =		M (1, 0) * M (0, 1)	1-0-1
	+	M (1, 1) * M (1, 1)	1-1-1
	+	M (1, 2) * M (2, 1)	1-2-1
	+	M (1, 3) * M (3, 1)	1-3-1

Il existe un chemin de longueur 2 pour aller de 1 à 1 si :

- il existe un chemin de 1 à 0 et de 0 à 1
- ou s'il existe un chemin de 1 à 2 et de 2 à 1
- ou s'il existe un chemin de 1 à 3 et de 3 à 1

M (1, 1) vrai correspondrait à une boucle sur le sommet 1, cas non envisagé dans ce chapitre.

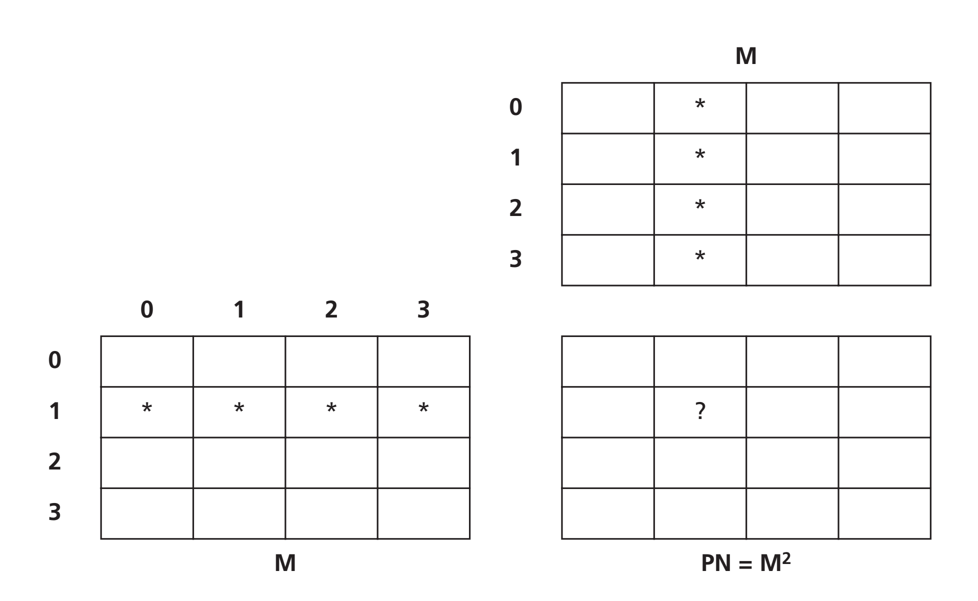


Figure 160 Produit de matrices.

L'exemple de la Figure 149 peut être représenté sous forme d'une matrice d'entiers, ou sous forme d'une matrice de booléens.

```
Nombre de chemins de longueur = 1
  0  1  0  0  0  0  1  0          F  V  F  F  F  F  V  F
  0  0  1  1  0  1  0  0          F  F  V  V  F  V  F  F
  0  0  0  1  0  0  0  0          F  F  F  V  F  F  F  F
  0  1  0  0  0  0  0  0          F  V  F  F  F  F  F  F
  0  0  0  1  0  1  0  1          F  F  F  V  F  V  F  V
  0  1  0  1  0  0  0  0          F  V  F  V  F  F  F  F
  0  0  0  0  0  1  0  0          F  F  F  F  F  V  F  F
  0  0  0  0  0  0  0  0          F  F  F  F  F  F  F  F
```

M² = M*M représente le nombre de chemins de longueur 2 si la matrice **M²** est entière, et l'existence d'un chemin de longueur 2 si **M²** est booléenne.

```
Nombre de chemins de longueur = 2
  0  0  1  1  0  2  0  0          F  F  V  V  F  V  F  F
  0  2  0  2  0  0  0  0          F  V  F  V  F  F  F  F
  0  1  0  0  0  0  0  0          F  V  F  F  F  F  F  F
  0  0  1  1  0  1  0  0          F  F  V  V  F  V  F  F
  0  2  0  1  0  0  0  0          F  V  F  V  F  F  F  F
  0  1  1  1  0  1  0  0          F  V  V  V  F  V  F  F
  0  1  0  1  0  0  0  0          F  V  F  V  F  F  F  F
  0  0  0  0  0  0  0  0          F  F  F  F  F  F  F  F
```

M^2 indique 2 chemins de longueur 2 de :

S0 à S5 : S0-S1-S5 et S0-S6-S5
S1 à S1 : S1-S3-S1 et S1-S5-S1
S1 à S3 : S1-S2-S3 et S1-S5-S3
S4 à S1 : S4-S3-S1 et S4-S5-S1

La somme SM = M + M^2 + M^3 ... + M^8 indique pour chaque i et j, le nombre de chemins de longueur <= 8 (produit entier), ou l'existence d'un chemin de longueur <= 8 (produit booléen) entre i et j.

La matrice booléenne SM représente la *fermeture transitive* :

```
F  V  V  V  F  V  V  F
F  V  V  V  F  V  F  F
F  V  V  V  F  V  F  F
F  V  V  V  F  V  F  F
F  V  V  V  F  V  F  V
F  V  V  V  F  V  F  F
F  V  V  V  F  V  F  F
F  F  F  F  F  F  F  F
```

On voit que pour le graphe de cet exemple, de S0, on peut aller vers S1, S2, S3, S5 et S6 (première ligne de la matrice). De S0, on ne peut pas aller vers S0, S4 ou S7. La dernière ligne indique que de S7, on ne peut aller nulle part.

> ***Remarque*** : un autre algorithme plus performant de calcul de la fermeture transitive existe connu sous le nom d'algorithme de Warshall. Cet algorithme envisage comme l'algorithme de Floyd un passage par les sommets intermédiaires k (de 0 à N-1) pour chaque couple w(i, j) de la matrice. Un chemin existe entre i et j, s'il existe déjà (w(i, j) est à vrai), ou s'il y a un chemin entre w(i, k) et w(k, j).

```
void warshall (GrapheMat* graphe) {
  int     ns   = graphe->n;
  int     nMax = graphe->nMax;
  Matrice w    = (int*) malloc (sizeof(int)*nMax*nMax);

  affMat (w, graphe->element, ns, nMax);

  for (int k=0; k<ns; k++) {
    for (int i=0; i<ns; i++) {
      for (int j=0; j<ns; j++) {
        w [i*nMax+j] = w [i*nMax+j] ||
                       w [i*nMax+k] && w [k*nMax+j];
      }
    }
  }

  ecrireMat (w, ns, nMax);
  free (w);
}
```

Remarque : on retrouve les résultats de l'algorithme de Floyd (voir *Passage par le sommet 6* § 5.6.8, page 288). La fermeture transitive s'identifie à la matrice de coût A des plus courts chemins en remplaçant INFINI ('*') par faux, et toutes les autres valeurs par vrai. Il y a une différence pour les éléments de la diagonale car dans l'algorithme de Floyd, on n'a pas retenu les plus courts chemins pour aller de i à i ; les éléments de la diagonale ont été initialisés à 0. Si on veut obtenir les plus courts chemins de longueur différente de 0 pour aller de i à i, il faut initialiser la diagonale de la matrice A à INFINI et non à 0.

Exercice 28 – Fermeture transitive par somme de produits de matrices

Écrire une fonction *void* ***produitEtFermeture*** *(GrapheMat* graphe) ;* qui à partir d'un graphe écrit la fermeture transitive du graphe calculée par somme de produits comme indiqué précédemment. Décomposer le problème en plusieurs fonctions réalisant le produit et la somme de matrices.

5.6.11 Menu de test des graphes (matrices)

Le menu de test donné pour la mémorisation sous forme de listes d'adjacence (voir § 5.5.10, page 276) reste valable (aux ifdef près), les prototypes des fonctions concernant les graphes sont les mêmes dans les deux cas (listes d'adjacence ou matrice) : *ajouterUnSommet()*, *ajouterUnArc()*, *parcoursProfond()*, etc.). Le calcul du plus court chemin est réalisé avec la fonction *void plusCourt (Graphe* graphe, int nsi) ;* dans le cas des listes d'adjacences et par la fonction *void floyd (GrapheMat* graphe) ;* pour la mémorisation sous forme de matrice. La création du graphe se fait également avec une fonction au prototype légèrement différent (valeur de retour).

5.7 RÉSUMÉ

Les graphes sont des structures de données très générales. On peut dire qu'un arbre est un cas particulier de graphe, et qu'une liste est un cas particulier d'arbre. Il existe de nombreux algorithmes sur les graphes. Ceux-ci sont examinés dans ce chapitre en mettant l'accent sur la mémorisation des graphes, l'allocation dynamique, la modularité et la récursivité.

Nous avons vu deux mémorisations possibles des graphes. La première représentation sous forme de listes d'adjacence convient lorsqu'il y a de nombreux sommets et peu de relations entre ces sommets. Une représentation sous forme de matrices conduirait à une matrice creuse. La mémorisation sous forme de matrices se fait en utilisant l'allocation dynamique pour allouer l'espace des matrices et ainsi éviter de devoir figer un maximum de sommets pour un graphe. Ceci nous a amenés à créer un nouveau type abstrait de données Graphe et quelques fonctions classiques sur les

graphes. Il existe de nombreux algorithmes concernant les graphes et le type Graphe pourrait facilement être enrichi de nouveaux prototypes de fonctions.

Les parcours de graphes en profondeur et en largeur sont une généralisation des parcours d'arbres. Il faut cependant veiller à marquer les sommets visités de façon à ne pas énumérer plusieurs fois le même sommet et à ne pas tourner en rond lorsqu'il y a présence de cycles.

Un problème classique des graphes consiste à trouver le plus court chemin d'un sommet vers les autres sommets. Le calcul de la fermeture transitive permet de connaître l'existence ou non (booléen) d'un chemin de longueur quelconque entre les couples de sommets.

5.8 CONCLUSION GÉNÉRALE

Les principales structures de données ont été présentées (listes, arbres, tables, graphes) en précisant à chaque fois leurs mémorisations, les algorithmes de parcours (énumération des éléments de la structure), de création et de désallocation, plus des algorithmes spécifiques à chaque structure de données.

Les nombreux exemples traités ont montré que ces différentes structures de données peuvent se combiner entre elles afin de résoudre ou d'optimiser tel ou tel aspect d'un programme. Le choix de la bonne structure de données est important, mais n'est pas toujours évident car il dépend de nombreux critères (nombre d'éléments, optimisation du temps d'accès ou de l'espace occupé, fréquence des ajouts, des retraits et des recherches, mémorisation en mémoire centrale ou secondaire, etc.). Il convient cependant dans la mesure du possible de définir des modules réutilisables. Le module de la gestion des listes par exemple est utilisé à maintes reprises dans les différents chapitres sans devoir replonger à chaque nouvelle application dans les détails de l'implémentation.

Corrigés des exercices

Exercice 1 : boucle sous forme récursive (page 8)

```
// boucle croissante de d à f, par pas de i (d:début, f:fin)
void boucleCroissante (int d, int f, int i) {
  if (d <= f) {
    printf ("Boucle valeur de n : %d\n", d);
    boucleCroissante (d+i, f, i);
  }
}
```

Exercice 2 : Hanoi : calcul du nombre de secondes ou d'années pour déplacer n disques (page 15)

```
/* dureehanoi.cpp  */

#include <stdio.h>

// 1 mouvement = 60 nanosecondes
// écrire le nombre de disques
// et le nombre de secondes pour déplacer les disques
void ecrireNbs (int n, double nbMvt) {
  double dureeEnSecondes =  (nbMvt * 60) / 1000000000;
  printf ("nb de disques et de secondes   : %2d : %.0f\n",
          n, dureeEnSecondes);
}

void main () {

  // nombre de mouvements pour n disques : 2 à la puissance n
  double nbMvt26 = 1 << 26;  // nombre de mouvements pour 26 disques
  double nbMvt27 = 1 << 27;
  double nbMvt28 = 1 << 28;
  double nbMvt29 = 1 << 29;
  double nbMvt30 = 1 << 30;
  double nbMvt31 = (double) 128.*256.*256.*256.;
  double nbMvt32 = (double) 256.*256.*256.*256.;
  double nbMvt64 = (double) 256.*256.*256.*256.*256.*256.*256.*256.;

  ecrireNbs (26, nbMvt26);
  ecrireNbs (27, nbMvt27);
  ecrireNbs (28, nbMvt28);
  ecrireNbs (29, nbMvt29);
  ecrireNbs (30, nbMvt30);
  ecrireNbs (31, nbMvt31);
  ecrireNbs (32, nbMvt32);


  // pour 64 disques
  double nsParAn = (3600 * 24. * 365.); // nombre de secondes par an
```

```
  double dureeEnSecondes =  (nbMvt64 * 60) / 1000000000;
  printf ("Nombre d'années pour %.0f mouvements :     %.0f\n",
                                        nbMvt64, dureeEnSecondes / nsParAn );
}


// n                  26 27 28 29 30  31  32
// durée en secondes   4  8 16 32 64 129 258
```

Exercice 3 : dessin d'un arbre (page 19)

Ajouter un paramètre à la fonction `avance()` permettant de spécifier la couleur du trait : `void avance (int lg, int x, int y, int angle, int* nx, int* ny, QColor couleur) ;`

```
// dessiner un segment et recommencer récursivement à partir de la fin
// du segment courant en éclatant le segment en nb autres segments
void dessinArbre (int lg, int x, int y, int angle, QColor couleur) {
  int nx, ny;
  avance (lg, x, y, angle, &nx, &ny, couleur);
  lg = 2*lg / 3;

  couleur = black;
  if (lg > 3) {
    if (lg <= 5) {
      couleur = green;      // feuille;
      int n2 = aleat (30);
      if (n2 == 1) {
        couleur = red;
      }
    }

    // ouverture de l'ensemble des nouveaux segments
    // pc : pourcentage d'aléatoire
    int nb   = 3 + aleat (3);
    int a    = angle;
    int d    = ouverture / nb;
    int douv = 1 + (int) (ouverture*pc);  // aléatoire ouverture
    int dlg  = 1 + (int) (lg*pc);         // aléatoire lg

    for (int i=1; i<=nb; i++) {
      a = angle - (ouverture/2) - (d/2) + i*d;
      a = a % 360;
      int deltaOuv = -(douv/2) + aleat (douv);
      int deltaLg  = aleat (dlg);
      dessinArbre (lg + deltaLg, nx, ny, a + deltaOuv, couleur);
    }
  }
}
```

Exercice 4 : spirale rectangulaire (récursive) (page 30)

```
/* spirale.cpp */

#include <stdio.h>
#include "ecran.h"

// exécute une boucle de la spirale à chaque appel;
// la longueur du segment tracé croît jusqu'à lgMax
void spirale (int n, int lgMax) {
  if (n < lgMax) {
    avancer (DROITE, n);
    avancer (HAUT  , n+1);
    avancer (GAUCHE, n+2);
    avancer (BAS,    n+3);
    spirale (n+4,    lgMax);
  }
}

void main () {

  initialiserEcran (20, 50);

  couleurCrayon (BLANC);
  crayonEn (10, 25);
  spirale ( 3, 15);
  afficherEcran ();

  detruireEcran ();
}
```

Exercice 5 : module de gestion de piles d'entiers (allocation contiguë) (page 30)

```
/* pile.cpp */

#include <stdio.h>
#include <stdlib.h>
#include "pile.h"

#define FATAL 1
#define AVERT 2

static void erreur (int typErr, char* message) {
  printf ("***Erreur : %s\n", message);
  if (typErr == FATAL) exit(0);
}

// le constructeur de Pile
Pile* creerPile (int max) {
  Pile* p    = (Pile*) malloc (sizeof (Pile));  // Pile* p = new Pile();
  p->max     = max;
  p->nb      = -1;
  p->element = (int*) malloc (max*sizeof(int));
  return p;
}

// la pile est-elle vide ?
int pileVide (Pile* p) {
  return p->nb == -1;
}

// empiler une valeur s'il reste de la place
void empiler (Pile* p, int valeur) {
  if (p->nb < p->max-1) {
    p->nb++;
    p->element [p->nb] = valeur;
  }  else {
    erreur (AVERT, "Pile saturée");
  }
}

// dépiler une valeur si la pile n'est pas vide
int depiler (Pile* p, int* valeur) {
  if ( !pileVide (p) ) {
    *valeur = p->element [p->nb];
    p->nb--;
    return 1;
  }  else {
    erreur (AVERT, "Pile vide");
    return 0;
  }
}

// lister les éléments de la pile
void listerPile (Pile* p) {
  if (pileVide (p) ) {
    printf ("Pile vide\n");
  } else {
    for (int i=0; i<=p->nb; i++) {
      printf ("%d ", p->element[i]);
    }
  }
}

// restituer l'espace allouée
void detruirePile (Pile* p) {
  free (p->element);
  free (p);
}
```

le programme principal de test :

```
/* pppile.cpp  programme principal des piles */

#include <stdio.h>
#include "pile.h"

#define faux        0
```

```
#define vrai        1
typedef int         booleen;

int menu (void) {
  printf ("\n\nGESTION D'UNE PILE\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Création    de la pile\n");
  printf ("2 - La pile est-elle vide ?\n");
  printf ("3 - Insertion   dans la pile\n");
  printf ("4 - Retrait     de   la pile\n");
  printf ("5 - Listage     de   la pile\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod);
  printf ("\n");

  return cod;
}

void main () {
  int taillePile;

  printf ("Taille de la pile d'entiers ? ");
  scanf  ("%d", &taillePile);
  Pile* p1 = creerPile (taillePile);

  booleen fini = faux;
  while (!fini) {

    switch (menu () ) {

    case 0 :
      fini = vrai;
      break;

    case 1 :
      detruirePile (p1);
      printf ("Taille de la pile d'entiers ? ");
      scanf  ("%d", &taillePile);
      p1 = creerPile (taillePile);
      break;

    case 2 :
      if (pileVide (p1) ) {
        printf ("Pile vide\n");
      } else {
        printf ("Pile non vide\n");
      }
      break;

    case 3 : {
      int valeur;
      printf  ("Valeur à empiler ? ");
      scanf   ("%d", &valeur);
      empiler (p1,   valeur);
      } break;

    case 4 : {
      int valeur;
      if (depiler (p1, &valeur)) {
        printf ("%d\n", valeur);
      } else {
        printf ("Pile vide\n");
      }
      } break;

    case 5 :
      listerPile (p1);
      break;
    } // switch
  }  // while

  detruirePile (p1);
}  // main
```

Exercice 6 : module de gestion de nombres complexes (page 32)

```
/* complex.cpp
   Module des opérations de base sur les complexes */

#include <stdio.h>
#include <math.h>   // cos, sin
#include "complex.h"

// création d'un Complexe
// à partir de partReel (partie réelle) et partImag (partie imaginaire)
Complex crC (double partReel, double partImag) {
  Complex z;
  z.partReel = partReel;
  z.partImag = partImag;
  return z;
}

// création d'un complexe à partir de ses Composantes en Polaire
Complex crCP (double module, double argument) {
  return crC (module * cos (argument), module * sin (argument));
}

// partie Réelle d'un Complexe
double partReelC (Complex z) {
  return z.partReel;
}

// partie Imaginaire d'un Complexe
double partImagC (Complex z) {
  return z.partImag;
}

// module d'un nombre Complexe
double moduleC (Complex z) {
  return sqrt (z.partReel * z.partReel + z.partImag * z.partImag);
}

// argument d'un nombre Complexe
double argumentC (Complex z) {
  return atan2 (z.partImag, z.partReel);
}

// écriture d'un nombre Complexe
void ecritureC (Complex z) {
  printf (" ( %5.2f + %5.2f i ) ", z.partReel, z.partImag);
}

// écriture en polaire d'un nombre Complexe
void ecritureCP (Complex z) {
  printf (" ( %5.2f, %5.2f) ", moduleC (z), argumentC (z));
}

// opposé d'un nombre Complexe
Complex opposeC (Complex z) {
  return crC (-z.partReel, -z.partImag);
}

// conjugué d'un nombre Complexe
Complex conjugueC (Complex z) {
  return crC (z.partReel, -z.partImag);
}

// inverse d'un nombre Complexe
Complex inverseC (Complex z) {
  return crCP (1/moduleC(z), -argumentC(z));
}

// puissance nième de z (n entier >=0)
Complex puissanceC (Complex z, int n) {
  Complex p = crC (1.0, 0.0);  // p = puissance
  for (int i=1; i<=n; i++) p = multiplicationC (p, z);
  return p;
}

// addition z = z1 + z
Complex additionC (Complex z1, Complex z2) {
  return crC (z1.partReel + z2.partReel, z1.partImag + z2.partImag);
}
```

```
// soustraction  z = z1 - z2
Complex soustractionC (Complex z1, Complex z2) {
  return crC (z1.partReel - z2.partReel, z1.partImag - z2.partImag);
}

// multiplication z = z1 * z2
Complex multiplicationC (Complex z1, Complex z2) {
  return crCP ( moduleC(z1) * moduleC(z2), argumentC(z1) + argumentC(z2) );
}

// division z = z1 / z2
Complex divisionC (Complex z1, Complex z2) {
  return multiplicationC (z1, inverseC (z2) );
}
```

le programme principal de test :

```
/* ppcomplex.cpp
   programme principal sur les opérations complexes */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>   // exit

#include "complex.h"

void ecrit (Complex z) {
  ecritureC  (z);
  ecritureCP (z);
}

void main() {
  Complex  c1, c2, c3, c4, c5, c6, c7;
  Complex  p1, p2, p3, p4, p5, p6, p7;

  c1 = crC (2,  1.50);
  c2 = crC (-2, 1.75);
  printf ("\nc1 =         "); ecrit (c1);
  printf ("\nc2 =         "); ecrit (c2);

  c3 = additionC       (c1, c2);
  c4 = soustractionC   (c1, c2);
  c5 = multiplicationC (c1, c2);
  c6 = divisionC       (c1, c2);
  c7 = puissanceC      (c1, 3);

  printf ("\nc3 = c1 + c2 "); ecrit (c3);
  printf ("\nc4 = c1 - c2 "); ecrit (c4);
  printf ("\nc5 = c1 * c2 "); ecrit (c5);
  printf ("\nc6 = c1 / c2 "); ecrit (c6);
  printf ("\nc7 = c1 ** 3 "); ecrit (c7);


  // en polaire
  printf ("\n\n");
  p1 = crCP (1, M_PI/4);
  p2 = crCP (1, M_PI/2);
  printf ("\np1 =         "); ecrit (p1);
  printf ("\np2 =         "); ecrit (p2);
  p3 = additionC       (p1, p2);
  p4 = soustractionC   (p1, p2);
  p5 = multiplicationC (p1, p2);
  p6 = divisionC       (p1, p2);
  p7 = puissanceC      (p1, 3);

  printf ("\np3 = p1 + p2 "); ecrit (p3);
  printf ("\np4 = p1 - p2 "); ecrit (p4);
  printf ("\np5 = p1 * p2 "); ecrit (p5);
  printf ("\np6 = p1 / p2 "); ecrit (p6);
  printf ("\np7 = p1 ** 3 "); ecrit (p7);

  printf ("\nPartie réelle de p1 : %.2f ", partReelC (p1));
  printf ("\nPartie imag.  de p1 : %.2f ", partImagC (p1));
  printf ("\nModule        de p1 : %.2f ", moduleC   (p1));
  printf ("\n");
}
```

Exercice 7 : polynômes d'une variable réelle (lecture, addition) (page 60)

```
/* polynome2.cpp utilise polynome.cpp */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>   // fabs
#include "polynome2.h"

#define ADD   1
#define SOUS -1

// lire le coefficient et l'exposant dans le fichier fe
static int lireMonome (FILE* fe, double* coefficient, int* puissance) {
  // fscanf fournit -1 en cas de eof : man fscanf
  int n = fscanf (fe, "%lf %d", coefficient, puissance);
  if (n > 0) {
    return 1;
  } else {
    return 0;
  }
}

// lire les valeurs de chaque monôme du polynôme p
Polynome* lirePolynome (FILE* fe) {
  Polynome* p = creerPolynome();
  double coefficient;
  int    puissance;
  while (lireMonome (fe, &coefficient, &puissance)) {
    Monome* nouveau = creerMonome (coefficient, puissance);
    insererEnOrdre (p, nouveau);
  }
  return p;
}

// addition     si typAouS vaut ADD,
// soustraction si typAouS vaut SOUS
// des polynômes a et b; résultat dans le polynôme c
static Polynome* addOuSousPoly (Polynome* a, Polynome* b, int typAouS) {
  Polynome* c = creerPolynome();

  ouvrirListe (a);
  // pa : pointeur courant sur le polynome a
  Monome* pa = (Monome*) objetCourant (a);
  ouvrirListe (b);
  // pb : pointeur courant sur le polynome b
  Monome* pb = (Monome*) objetCourant (b);

  // tant qu'on n'a pas atteint la fin du polynôme a
  // ni celle du polynôme b
  while (pa!=NULL && pb!=NULL) {
    if (pa->exposant == pb->exposant) {
      double s = pa->coefficient + typAouS * pb->coefficient;
      if (s != 0) {
        Monome* nouveau = creerMonome (s, pa->exposant);
        insererEnOrdre (c, nouveau);
      }
      pa = (Monome*) objetCourant (a);
      pb = (Monome*) objetCourant (b);

    } else  {
      if (pa->exposant < pb->exposant) {
        Monome* nouveau = creerMonome (typAouS*pb->coefficient, pb->exposant);
        insererEnOrdre (c, nouveau);
        pb = (Monome*) objetCourant (b);
      } else {
        Monome* nouveau = creerMonome (pa->coefficient, pa->exposant);
        insererEnOrdre (c, nouveau);
        pa = (Monome*) objetCourant (a);
      }
    }
  }

  // recopier la fin du polynôme a
  while (pa!=NULL) {
    Monome* nouveau = creerMonome (pa->coefficient, pa->exposant);
    insererEnOrdre (c, nouveau);
    pa = (Monome*) objetCourant (a);
```

```
  }

  // recopier la fin du polynôme b
  while (pb!=NULL) {
    Monome* nouveau = creerMonome (typAouS*pb->coefficient, pb->exposant);
    insererEnOrdre (c, nouveau);
    pb = (Monome*) objetCourant (b);
  }

  return c;
}

Polynome* addPolynome (Polynome* a, Polynome* b) {
  return addOuSousPoly (a, b, ADD);
}

Polynome* sousPolynome (Polynome* a, Polynome* b) {
  return addOuSousPoly (a, b, SOUS);
}
```

le programme principal de test :

```
/* pppolynome.cpp */

#include <stdio.h>
#include <stdlib.h>
#include "polynome2.h"

void main () {
  // ouverture des fichiers
  FILE* fe1 = fopen ("polya.dat", "r");
  if (fe1==NULL)  {
    fprintf (stderr, "fe1 fichier inconnu\n"); exit (0);
  }
  FILE* fe2 = fopen ("polyb.dat", "r");
  if (fe2==NULL)  {
    fprintf (stderr, "fe2 fichier inconnu\n"); exit (0);
  }

  Polynome* a = lirePolynome (fe1);
  Polynome* b = lirePolynome (fe2);
  Polynome* c = addPolynome  (a, b);
  Polynome* d = sousPolynome (a, b);

  printf ("\nPolynome a           : "); listerPolynome (a);
  printf ("\nPolynome b           : "); listerPolynome (b);
  printf ("\nPolynome c = a + b   : "); listerPolynome (c);
  printf ("\nPolynome d = a - b   : "); listerPolynome (d);

  printf ("\nValeur de a pour x=1 : %14.2f", valeurPolynome (a, 1) );
  printf ("\nValeur de b pour x=1 : %14.2f", valeurPolynome (b, 1) );
  printf ("\nValeur de c pour x=1 : %14.2f", valeurPolynome (c, 1) );
  printf ("\nValeur de d pour x=1 : %14.2f", valeurPolynome (d, 1) );
  printf ("\n");
}
```

exemples de fichiers décrivant des polynômes :

```
/* polya.dat */
-6  7
3   5
-1  1
/* polyb.dat */
6   7
3   6
5   5
-2  3
3   1
5   0
```

exemple de résultats :

```
Polynome a           :  -6.00 x**7  +3.00 x**5  -1.00 x**1
Polynome b           :  +6.00 x**7  +3.00 x**6  +5.00 x**5  -2.00 x**3
                                                         +3.00 x**1  +5.00 x**0
Polynome c = a + b   :  +3.00 x**6  +8.00 x**5  -2.00 x**3  +2.00 x**1
                                                                  +5.00 x**0
Polynome d = a - b   :  -12.00 x**7  -3.00 x**6  -2.00 x**5  +2.00 x**3
                                                          -4.00 x**1  -5.00 x**0
Valeur de a pour x=1 :          -4.00
Valeur de b pour x=1 :          20.00
Valeur de c pour x=1 :          16.00
Valeur de d pour x=1 :         -24.00
```

Exercice 8 : systèmes experts : les algorithmes de déduction (page 66)

```
// le fait num existe-t'il dans la liste listF
booleen  existe (ListeFaits* listF, int num) {
  booleen trouve = faux;
  ouvrirListe (listF);
  while (!finListe (listF) && !trouve) {
    Fait* ptc    = (Fait*) objetCourant (listF);
    trouve = ptc->numero == num;
  }
  return trouve;
}

// appliquer la règle pointée par regle à la liste de faits listF
int appliquer (Regle* regle, ListeFaits* listF) {

  // Les hypothèses sont-elles vérifiées
  booleen verifie = vrai;  // hypothèses vraies a priori
  Liste*  lh = regle->hypotheses;
  ouvrirListe (lh);
  while (!finListe (lh) && verifie) {
    Fait* ptc = (Fait*) objetCourant (lh);
    verifie   = existe (listF, ptc->numero);
  }

  // toutes les hypothèses de regle sont vraies;
  // il faut ajouter les conclusions de la règle à la liste de faits listF
  if (verifie) {
    regle->marque = vrai;  // une règle ne s'applique qu'une fois
    Liste* lc = regle->conclusions;
    ouvrirListe (lc);
    while (!finListe (lc)) {
      Fait* ptc = (Fait*) objetCourant (lc);
      if (!existe (listF, ptc->numero)) {
        ajouterFait (listF, ptc->numero);
      }
    }
  }
  return verifie;
}

// À partir de la liste de faits, on essaie d'appliquer successivement
// toutes les règles. Si une règle s'est appliquée, de nouveaux faits
// ont été ajoutés qui peuvent permettre à des règles déjà examinées
// de s'appliquer.
// Si au moins une règle s'est appliquée, on refait un tour complet.
// Une règle ne s'applique qu'une seule fois.
void chainageAvant (ListeRegles* listR, ListeFaits* listF) {
  int fini = faux;

  while (!fini) {
    booleen again = faux;  // refaire un examen des règles
                           // si au moins une règle s'applique
    ouvrirListe (listR);
    while (!finListe (listR) ) {
      Regle* ptc = (Regle*) objetCourant (listR);
      // si la règle n'a pas déjà été exécutée,
      // on essaie de l'appliquer.
      if (!ptc->marque) {
        int resu = appliquer (ptc, listF);
        if (resu) again = vrai;
      }
    }
    if (!again) fini = vrai;
  }
}

// écrire nb espaces
void indenter (int nb) {
  for (int i=0; i<nb; i++) printf (" ");
}

// à partir de la liste des règles et de la liste des faits,
// démontrer le fait num.
// nb sert à faire une indentation des résultats
booleen demontrerFait (ListeRegles* listR, ListeFaits* listF, int num, int nb) {
  booleen ok;
  indenter (nb); printf ("demontreFait %d\n", num);
```

```
  if ( existe (listF, num) ) {  // le fait est dans la liste de faits
    indenter (nb); printf ("Dans Liste de Faits %d\n", num);
    ok = vrai;

  } else { // rechercher une règle ayant num en conclusions
    Regle* ptr;
    ouvrirListe (listR);
    booleen trouve = faux;
    while (!finListe (listR) && !trouve ) {
      ptr    = (Regle*) objetCourant (listR);
      trouve = existe (ptr->conclusions, num);
    }

    if (!trouve) {
      indenter (nb); printf ("Pas démontré\n");
      ok = faux;
    } else {
      indenter (nb); printf ("Voir règle %s\n", ptr->nom);

      // démontrer récursivement chacune des hypothèses de la règle ptr
      Liste* lh = ptr->hypotheses; // liste des hypothèses de la règle ptr
      ouvrirListe (lh);
      ok = vrai;  // a priori
      while (!finListe(lh) && ok ) {
        Fait* ptf = (Fait*) objetCourant (lh);
        ok = demontrerFait (listR, listF, ptf->numero, nb+10);
      }
    }
  }  // if
  return ok;
}
```

Le programme principal :

```
char* message (int n) {
#if 0
static char* libelle [] = {
" ",
"allaite ses petits",
"a des dents pointues",
"vit en compagnie de l'homme",
"grimpe aux arbres",
"a des griffes pointues",
"est domestiqué",
"est couvert de poils",
"a quatre pattes",
"est un mammifère",
"est un carnivore",
"est un chat"
} ;
#else
static char* libelle [] = {
" ",
"a de la fièvre",
"a le nez bouché",
"a mal au ventre",
"a des frissons",
"a la gorge rouge",
"a l'appendicite",
"a mal aux oreilles",
"a mal à la gorge",
"a les oreillons",
"a un rhume",
"a la grippe"
};
#endif

 return libelle [n];
}

void main () {
  ListeFaits* listF = creerListeFaits();

  ajouterFait (listF, 1);
  ajouterFait (listF, 2);
  ajouterFait (listF, 3);
  ajouterFait (listF, 4);
  ajouterFait (listF, 5);

  printf ("Liste faits : \n");
```

```
  listerFaits (listF);

  ListeRegles* listR = creerListe();

  Regle* PRA;

  PRA = creerRegle    ("A");
  ajouterHypothese    (PRA, 3);
  ajouterConclusion   (PRA, 6);
  insererEnFinDeListe (listR, PRA);

  PRA = creerRegle    ("B");
  ajouterHypothese    (PRA, 1);
  ajouterHypothese    (PRA, 7);
  ajouterHypothese    (PRA, 8);
  ajouterConclusion   (PRA, 9);
  insererEnFinDeListe (listR, PRA);

  PRA = creerRegle    ("C");
  ajouterHypothese    (PRA, 5);
  ajouterHypothese    (PRA, 10);
  ajouterConclusion   (PRA, 11);
  insererEnFinDeListe (listR, PRA);

  PRA = creerRegle    ("D");
  ajouterHypothese    (PRA, 1);
  ajouterHypothese    (PRA, 2);
  ajouterConclusion   (PRA, 10);
  insererEnFinDeListe (listR, PRA);

  listerLesRegles (listR);

  // chaînage arrière
  booleen resu;
  resu = demontrerFait (listR, listF, 11, 0);
  printf ("Resu %d\n", resu);

  // chaînage avant : modifie la liste des faits listF
  chainageAvant (listR, listF);
  printf ("\nListe faits après chaînage avant : \n");
  listerFaits (listF);
}
```

Exercice 9 : le type pile (allocation contiguë) (page 72)

Le fichier d'en-tête piletableau.h pour les piles utilisant un tableau :

```
/* piletableau.h  version avec allocation dynamique du tableau */

#ifndef PILE_H
#define PILE_H

typedef int  booleen;
#define faux 0
#define vrai 1
typedef void Objet;

typedef struct {
  int     max;         // nombre maximum d'éléments dans la pile
  int     nb;          // repère le dernier occupé de element
  Objet** element;     // le tableau des éléments de la pile
} Pile;

Pile*  creerPile    (int   max);
int    pileVide     (Pile* p);
void   empiler      (Pile* p, Objet* objet);
Objet* depiler      (Pile* p);
void   listerPile   (Pile* p, void (*f) (Objet*));
void   detruirePile (Pile* p);

#endif
```

Le corps du module `piletableau.cpp` :

```
// piletableau.cpp

#include <stdio.h>
#include <stdlib.h>
#include "piletableau.h"
```

```
#define FATAL 1
#define AVERT 2

static void erreur (int typErr, char* message) {
  printf ("***Erreur : %s\n", message);
  if (typErr == FATAL) exit(0);
}

// le constructeur de Pile
Pile* creerPile (int max) {
  Pile* p    = (Pile*) malloc (sizeof (Pile));
  p->max     = max;
  p->nb      = -1;
  p->element = (Objet**) malloc (max*sizeof(Objet*));
  return p;
}

// la pile est-elle vide ?
int pileVide (Pile* p) {
  return p->nb == -1;
}

// empiler une valeur s'il reste de la place
void empiler (Pile* p, Objet* objet) {
  if (p->nb < p->max-1) {
    p->nb++;
    p->element [p->nb] = objet;
  }  else {
    erreur (AVERT, "Pile saturée");
  }
}

// dépiler une valeur si la pile n'est pas vide
Objet* depiler (Pile* p) {
  if ( !pileVide (p) ) {
    Objet* objet = p->element [p->nb];
    p->nb--;
    return objet;
  }  else {
    erreur (AVERT, "Pile vide");
    return NULL;
  }
}

// lister les éléments de la pile
void listerPile (Pile* p, void (*f) (Objet*)) {
  if (pileVide (p) ) {
    printf ("Pile vide\n");
  } else {
    for (int i=0; i<=p->nb; i++) {
      f (p->element[i]);
    }
  }
}

// restituer l'espace allouée
void detruirePile (Pile* p) {
  free (p->element);
  free (p);
}
```

Le programme principal est le même que pour la pile gérée avec une liste (voir § 2.4.5.e, page 69).

Exercice 10 : files d'attente en allocation dynamique (page 73)

```
// file.cpp  file gérée à l'aide d'une liste simple

#include <stdio.h>
#include <stdlib.h>

#include "file.h"

// créer et initialiser une File
File* creerFile () {
  return creerListe ();
}

// vrai si la File est vide, faux sinon
booleen fileVide (File* file) {
```

```
  return listeVide (file);
}

// ajouter un objet dans la File
void enFiler (File* file, Objet* objet) {
  insererEnFinDeListe (file, objet);
}

// fournir l'adresse de l'objet en tête de File
// ou NULL si la File est vide
Objet* deFiler (File* file) {
  if (fileVide (file)) {
    return NULL;
  } else {
    return extraireEnTeteDeListe (file);
  }
}

// lister les objets de la File
void listerFile (File* file, void (*f) (Objet*)) {
  listerListe (file, f);
}

// détruire une File
void detruireFile (File* file) {
  detruireListe (file);
}
```

le programme principal `ppfile.cpp` (la variable de compilation FILETABLEAU n'est pas définie pour cet exemple).

```
// ppfile.cpp  programme principal des files (avec listes ou avec tableaux)

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#if FILETABLEAU
#include "filetableau.h"
#else
#include "file.h"
#endif

#include "mdtypes.h"

int menu () {
  printf ("\n\nGESTION D'UNE FILE D'ENTIERS\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Initialisation de la file\n");
  printf ("2 - La file est-elle vide\n");
  printf ("3 - Insertion   dans la file\n");
  printf ("4 - Retrait     de   la file\n");
  printf ("5 - Listage     de   la file\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod);
  printf ("\n");

  return cod;
}

void main () {
  #if FILETABLEAU
  File*   file1 = creerFile(7);
  #else
  File*   file1 = creerFile();
  #endif
  booleen fini = faux;

  while (!fini) {

    switch (menu() ) {

    case 0 :
      fini = vrai;
      break;

    case 1 :
      detruireFile (file1);
```

```
      #if FILETABLEAU
      file1 = creerFile(7);
      #else
      file1 = creerFile();
      #endif
      break;

    case 2 :
      if (fileVide (file1) ) {
        printf ("File vide\n");
      } else {
        printf ("File non vide\n");
      }
      break;

    case 3 : {
      int valeur;
      printf  ("Valeur à enfiler ? ");
      scanf   ("%d", &valeur);
      enFiler (file1, creerEntier(valeur));
      } break;

    case 4 : {
      Entier* v;
      if ((v=(Entier*)deFiler (file1)) != NULL) {
        ecrireEntier (v);
      } else {
        printf ("File vide\n");
      }
      } break;

    case 5:
      listerFile (file1, ecrireEntier);
      break;
    }
  }

  detruireFile (file1);


  printf ("\n\nGESTION D'UNE FILE DE PERSONNES\n\n");
  #if FILETABLEAU
  File*   file2 = creerFile(7);
  #else
  File*   file2 = creerFile();
  #endif

  enFiler (file2, creerPersonne ("Dupont",   "Jacques"));
  enFiler (file2, creerPersonne ("Dufour",   "Michel"));
  enFiler (file2, creerPersonne ("Dupré",    "Jeanne"));
  enFiler (file2, creerPersonne ("Dumoulin", "Marie"));
  listerFile (file2, ecrirePersonne);
}
```

Exercice 11 : files d'attente en allocation contiguë (page 74)

```
/* filetableau.cpp */

#include <stdio.h>
#include <stdlib.h>

#include "filetableau.h"

#define FATAL 1
#define AVERT 2

static void erreur (int typErr, char* message) {
  printf ("***Erreur : %s\n", message);
  if (typErr == FATAL) exit (0);
}

File* creerFile (int max) {
  File* file    = new File();
  file->max     = max;
  file->premier = max-1;
  file->dernier = max-1;
  // allouer un tableau de pointeurs d'objets
  file->element = (Objet**) malloc (max * sizeof(Objet*));
```

```
  return file;
}

int fileVide (File* file) {
  return file->premier == file->dernier;
}

void enFiler (File* file, Objet* objet) {
  int place = (file->dernier+1) % file->max;
  if (place != file->premier) {
     file->dernier = place;
     file->element [place] = objet;
  }  else {
    erreur (AVERT, "File saturée");
  }
}

Objet* deFiler (File* file) {
  int place = (file->premier+1) % file->max;

  if (!fileVide (file)) {
     Objet* objet = file->element [place];
     file->premier = place;
     return objet;
  }  else {
    erreur (AVERT, "File vide");
    return NULL;
  }
}

void listerFile (File* file, void (*f) (Objet*)) {
  printf ("Premier : %d, Dernier : %d\n", file->premier, file->dernier);

  if (fileVide (file)) {
    printf ("File vide\n");
  } else {
    for (int i=(file->premier+1) % file->max;
         i!=(file->dernier+1) % file->max; i=(i+1) % file->max ) {
      f (file->element[i]);
    }
    printf ("\n");
  }
}

void detruireFile (File* file) {
  free (file->element);
}
```

`ppfile.cpp` est le même que pour l'exercice 10 sauf que `creerFile()` a un paramètre. La variable de compilation FILETABLEAU est définie pour cet exemple.

Exercice 12 : commande en attente (page 97)

```
// insertion en fin des listes de l'article et du client
// en attente de qt articles de numéro numArt
// pour le client numCli à la date "date"
void mettreEnAttente (int qt, int numArt, int numCli, char* date) {
  non corrigé
}


// extraire l'entrée n des listes de Attente
// et l'insérer dans la liste libre
void extraire (int n) {
  non corrigé
}

// réapprovisionnement de qtr articles de numéro na
void reappro (int na, int qtr) {
  Attente  attente;
  Article  article;
  Client   client;

  lireDArt (na, &article);
  if (article.premier != NILE) {
    int ptc = article.premier;
    booleen fini = faux;
    while ((ptc != NILE) && !fini) {
```

```
      lireDAtt (ptc, &attente);
      lireDCli (attente.cliOuLL, &client);
      if (attente.qt <= qtr) {
        printf ("Envoi de %2d %15s à %15s\n",
          attente.qt, article.nomArt, client.nomCli);
        qtr -= attente.qt;
        fini = qtr == 0;
        extraire (ptc);
        ptc = attente.artSuivant;
      } else {
        printf ("Envoi de %2d %15s à %15s (%2d commandé)\n",
          qtr, article.nomArt, client.nomCli, attente.qt);
        attente.qt -= qtr;
        ecrireDAtt (ptc, &attente);
        fini = vrai;
      }
    }
  }
}

void main () {
  fa = fopen ( "article.rel", "wb+");
  if (fa == NULL) { printf ("Erreur ouverture FA\n"); exit (1); }
  fc = fopen ( "client.rel",  "wb+");
  if (fc == NULL) { printf ("Erreur ouverture FC\n"); exit (1); }

  initArt ("Radio",        1,  0);
  initArt ("Téléviseur",   3,  0);
  initArt ("Magnétoscope", 8, 25);
  initArt ("Chaine hi-fi", 10, 0);

  initCli ("Dupond", 2);
  initCli ("Durand", 5);
  initCli ("Dufour", 6);

  initAtt ("attente.rel");

  mettreEnAttente ( 3,  3, 2, "03/07/..");
  mettreEnAttente ( 1,  1, 2, "10/08/..");
  mettreEnAttente ( 5, 10, 2, "02/09/..");
  mettreEnAttente (10,  3, 5, "12/09/..");
  mettreEnAttente ( 7, 10, 5, "13/09/..");

  listerTous ();
  listerArt (5);   // lister les articles du client 5
  listerCli (3);   // lister les clients pour l'article 3

  reappro (3, 10); // réapprovisionnement de 10 articles numéro 3

  listerTous ();
  listerArt (5);   // lister les articles du client 5
  listerCli (3);   // lister les clients pour l'article 3

  fclose    (fa);
  fclose    (fc);
  fermerAtt ();
}
```

Exercice 13 : les cartes à jouer (page 98)

```
/* cartes.cpp  module des cartes */

#include <stdio.h>
#include <stdlib.h>

#include "cartes.h"

void insererEnFinDePaquet (PaquetCarte* p, int couleur, int valeur) {
  Carte* carte = new Carte();
  if (carte==NULL) {
    printf ("Erreur allocation de Carte\n"); exit (0);
  }
  carte->couleur = couleur;
  carte->valeur  = valeur;
  insererEnFinDeListe (p, carte);
}

void listerCartes (PaquetCarte* p) {
```

```
  int i=0;
  ouvrirListe (p);
  while (!finListe (p)) {
    i++;
    Carte* carte = (Carte*) objetCourant (p);
    printf ("i %2d : C %2d, V %2d\n", i, carte->couleur, carte->valeur);
  }
}

void creerTas (PaquetCarte* p) {
  initListe (p);
  for (int i=1; i<=4; i++) {
    for (int j=1; j<=13; j++) insererEnFinDePaquet (p, i, j);
  }
}

// La première carte a le numéro 1;
// fournit NULL si la carte n n'existe pas
Carte* extraireNieme (PaquetCarte* p, int n) {
  Carte* extrait;
  if (n<=0 || n>nbElement (p)) return NULL;
  ouvrirListe (p);
  for (int i=1; i<=n; i++) extrait = (Carte*) objetCourant (p);
  int resu = extraireUnObjet (p, extrait);
  return extrait;
}

void battreLesCartes (PaquetCarte* p, PaquetCarte* paquetBattu) {
  initListe (paquetBattu);
  for (int i=1; i<=52; i++) {
    int n  = (rand() % (52-i+1)) + 1;
    Carte* extrait = extraireNieme (p, n);
    if (extrait != NULL) insererEnFinDeListe (paquetBattu, extrait);
  }
}

void distribuerLesCartes (PaquetCarte* p, tabJoueur joueur) {
  for (int nj=0; nj<4; nj++) initListe (&joueur[nj]);

  for (int i=1; i<=13; i++) {
    for (int nj=0; nj<4; nj++) {
      Carte* extrait = (Carte*) extraireEnTeteDeListe (p);
      insererEnFinDeListe (&joueur[nj], extrait);
    }
  }
}
```

le programme principal :

```
/* ppcartes.cpp  programme principal cartes */

#include <stdio.h>
#include <stdlib.h>
#include "cartes.h"

void main () {

  // créer un tas et le lister
  PaquetCarte* tas1 = new PaquetCarte();
  creerTas (tas1);
  printf   ("\nListe du tas1 au départ \n");
  listerCartes (tas1);

  // battre les cartes et afficher le nouveau tas tas2
  PaquetCarte* tas2 = new PaquetCarte();
  battreLesCartes (tas1, tas2);
  //printf       ("\nListe du tas1 après BattreLesCartes\n");
  //listerCartes (tas1);  // vide
  printf       ("\nListe du tas2 après BattreLesCartes\n");
  listerCartes (tas2);

  // distribuer les cartes et afficher les 4 paquets des joueurs
  tabJoueur    joueur;
  distribuerLesCartes (tas2, joueur);
  for (int i=0; i<4; i++) {
    printf       ("\nListe du joueur %d \n", i);
    listerCartes (&joueur[i]);
  }
}
```

Exercice 14 : polynômes complexes (listes ordonnées) (page 99)

```
/* polynome.cpp  gestion de polynomes complexes */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "polynome.h"

// comparer deux monômes
static int comparerMonome (Monome* m1, Monome* m2) {
  if (m1->puissance < m2->puissance) {
    return -1;
  } else if (m1->puissance == m2->puissance) {
    return 0;
  } else {
    return 1;
  }
}

static int comparerMonome (Objet* objet1, Objet* objet2) {
  return comparerMonome ((Monome*)objet1, (Monome*)objet2);
}

void creerMonome (Polynome* p, Complex z, int puissance) {
  Monome* nouveau    = new Monome;
  nouveau->z         = z;
  nouveau->puissance = puissance;
  insererEnOrdre (p, nouveau);
}

Polynome* creerPolynome () {
  return creerListe (DECROISSANT, NULL, comparerMonome);
}

Polynome* lirePolynome () {
  Polynome* p = creerPolynome();
  booleen fin = faux;
  while (!fin) {
    printf ("Coefficient réel (ou * pour finir) ? ");
    char ch [30];
    scanf ("%s", ch);
    if ( !strcmp (ch, "*") == 0 ) {
      double pr = atof (ch);
      printf ("Coefficient imaginaire ? ");
      scanf  ("%s", ch);
      double pi = atof (ch);
      Complex z = crC (pr, pi);

      printf ("Puissance ? ");
      scanf  ("%s", ch);
      int pu = atoi (ch);
      creerMonome (p, z, pu);
    } else {
      fin = vrai;
    }
  }
  return p;
}

void ecrirePolynome (Polynome* po) {
  ouvrirListe (po);
  while (!finListe (po) ) {
    Monome* ptc = (Monome*) objetCourant (po);
    Complex c   = ptc -> z;
    double  pr  = partReelC (c);
    double  pi  = partImagC (c);
    int     pu  = ptc->puissance;

    if ( pr != 0 && pi != 0 ) {
      printf ( " (%.2f + %.2f i) z** %d ", pr, pi, pu);
    } else if (pr != 0) {
      printf ( " %.2f z** %d ", pr, pu);
    } else {
      printf ( " %.2f i z** %d ", pi, pu);
    }
    if (!finListe (po) ) printf (" + ");
  }
  printf ("\n");
}
```

```
Complex valeurPolynome (Polynome* po, Complex z) {
  Complex resu = crC (0, 0);
  ouvrirListe (po);
  while (!finListe (po)) {
    Monome* ptc = (Monome*) objetCourant (po);
    Complex mc = multiplicationC (ptc->z, puissanceC (z, ptc->puissance));
    resu = additionC (resu, mc);
  }
  return resu;
}
```

et le programme principal de test :

```
/* pppolynomecomplex.cpp */

#include <stdio.h>
#include "polynome.h"

void main () {

  Polynome* p1 = creerPolynome();
  creerMonome    (p1, crC(2,2), 2);
  creerMonome    (p1, crC(1,1), 1);
  creerMonome    (p1, crC(3,3), 3);
  printf ("\n\nListe du polynome p1 : ");
  ecrirePolynome (p1);

  Polynome* p2 = creerPolynome();
  creerMonome    (p2, crC(1,1), 3);
  creerMonome    (p2, crC(2,2), 1);
  creerMonome    (p2, crC(3,3), 2);
  printf ("\nListe du polynome p2 : ");
  ecrirePolynome (p2);

  Complex z1 = crC (0.5, 1);
  Complex resu1 = valeurPolynome  (p1, z1);
  Complex resu2 = valeurPolynome  (p2, z1);
  Complex resum = multiplicationC (resu1, resu2);
  Complex resud = divisionC       (resu1, resu2);

  printf ("\nz1           = "); ecritureC (z1);
  printf ("\np1(z1)       = "); ecritureC (resu1);
  printf ("\np2(z1)       = "); ecritureC (resu2);
  printf ("\np1(z1)*p2(z1) = "); ecritureC (resum);
  printf ("\np1(z1)/p2(z1) = "); ecritureC (resud);
  printf ("\n");
}
```

Exercice 15 : parcours d'arbres droite-gauche (page 116)

```
Parcours préfixé : Julie Jonatan Gontran Antonine Pauline Sonia Paul

Parcours infixé : Julie Gontran Antonine Jonatan Paul Sonia Pauline

Parcours postfixé : Antonine Gontran Paul Sonia Pauline Jonatan Julie
```

Exercice 16 : dessin n-aire d'un arbre (page 136)

```
// dessiner arbre n-aire

// moyenne de la position du premier et dernier
// fils n-aires de racine
static int positionMoyenne (Noeud* racine) {
  Noeud* pf = racine->gauche;  // pointeur fils NAire
  int pos1 = ((NomPos*)pf->reference)->position;
  while (pf->droite != NULL) pf = pf->droite;
  int posD = ((NomPos*)pf->reference)->position;
  return (pos1+posD)/2;
}

// nf : nombre de feuilles n-aires
int nf = 0;  // variable globale pour dupArbN

// lgM = largeur d'une colonne
static Noeud* dupArbN (Noeud* racine, int lgM, char* (*toString) (Objet*) ) {
  if (racine == NULL) {
    return NULL;
  } else {
```

```
    Noeud* nouveau  = new Noeud();
    NomPos* objet   = new NomPos();
    nouveau->reference = objet;
    objet->message  = toString (racine->reference);
    nouveau->gauche = dupArbN (racine->gauche, lgM, toString);

    if (racine->gauche == NULL) {  // c'est une feuille n-aire
      ++nf;
      objet->position = lgM * nf;
    } else {
      objet->position = positionMoyenne (nouveau);
    }

    nouveau->droite   = dupArbN (racine->droite, lgM, toString);
    return nouveau;
  }
}

static Arbre* dupArbN (Arbre* arbre, int lgM) {
  nf  = 0;
  Noeud* nrac = dupArbN (arbre->racine, lgM, arbre->toString);
  return creerArbre (nrac, arbre->toString, NULL);
}

// dessiner l'arbre n-aire dans le fichier fs (à l'écran si fs=stdout)
void dessinerArbreNAire (Arbre* arbre, FILE* fs) {
  if (arbreVide (arbre)) {
    printf ("dessinerArbre  Arbre vide\n");
    return;
  }

  int lgM = maxIdent (arbre) + 1 ;
  if (lgM < 5) lgM = 5;
  int   lgFeuille = (nbFeuillesNAire (arbre)+1) * lgM;
  char* ligne     = (char*) malloc (lgFeuille+1);
  ligne [lgFeuille] = 0;
  Arbre* narbre = dupArbN (arbre, lgM);

  Liste* lc = creerListe();  // liste des noeuds du même niveau
  insererEnFinDeListe (lc, narbre->racine);
  Liste* ls = creerListe();  // liste des descendants de lc

  while (!listeVide (lc)) {
    // écrire les barres verticales des noeuds de la liste
    for (int i=0; i<lgFeuille; i++) ligne[i]=' ';
    ouvrirListe (lc);
    while (!finListe (lc) ) {
      Noeud* ptNd = (Noeud*) objetCourant (lc);
      NomPos* ptc = (NomPos*) ptNd->reference;
      ligne [ptc->position] = '|';
    }
    for (int i=1; i<=2; i++) fprintf (fs, "%s\n", ligne);

    // pour chaque élément de la liste :
    //   écrire des tirets de la position du premier
    //     fils à celle du dernier
    //   écrire le nom de l'élément à sa position
    for (int i=0; i<lgFeuille; i++) ligne[i]=' ';
    while (!listeVide (lc)) {
      Noeud* pNC = (Noeud*) extraireEnTeteDeListe (lc);
      Noeud* pF  = pNC->gauche;
      char*  message = ((NomPos*) pNC->reference)->message;
      int lg     = strlen (message);
      int posNom = ((NomPos*)pNC->reference)->position - lg/2;
      int pos1F;    // position du premier fils
      int posDF;    // position du dernier fils
      if (pF == NULL) {
        pos1F = posNom;
        posDF = posNom;
      } else {
        pos1F = ((NomPos*)pF->reference)->position;
        while (pF != NULL) {
          insererEnFinDeListe (ls, pF);
          posDF = ((NomPos*)pF->reference)->position;
          pF    = pF->droite;
        }
      }

      for (int j=pos1F; j<=posDF; j++) ligne [j] = '_';
```

```
      for (int j=0;     j<lg;     j++) ligne [posNom+j] = message[j];
    }

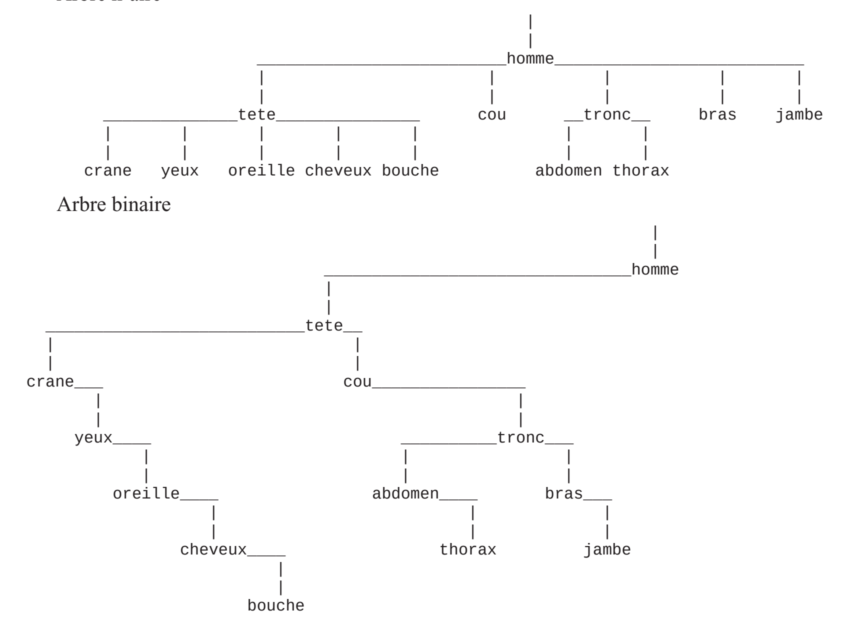
    fprintf (fs, "%s\n", ligne);
    recopierListe (lc, ls);   // ls vide
  }

  // détruire l'arbre intermédiaire
  detruireArbre (narbre);
}
```

Exercice 17 : le corps humain (page 142)

Seules les 3 premières lignes de homme.nai sont prises en compte pour cette correction qui reste à compléter.

Arbre n-aire

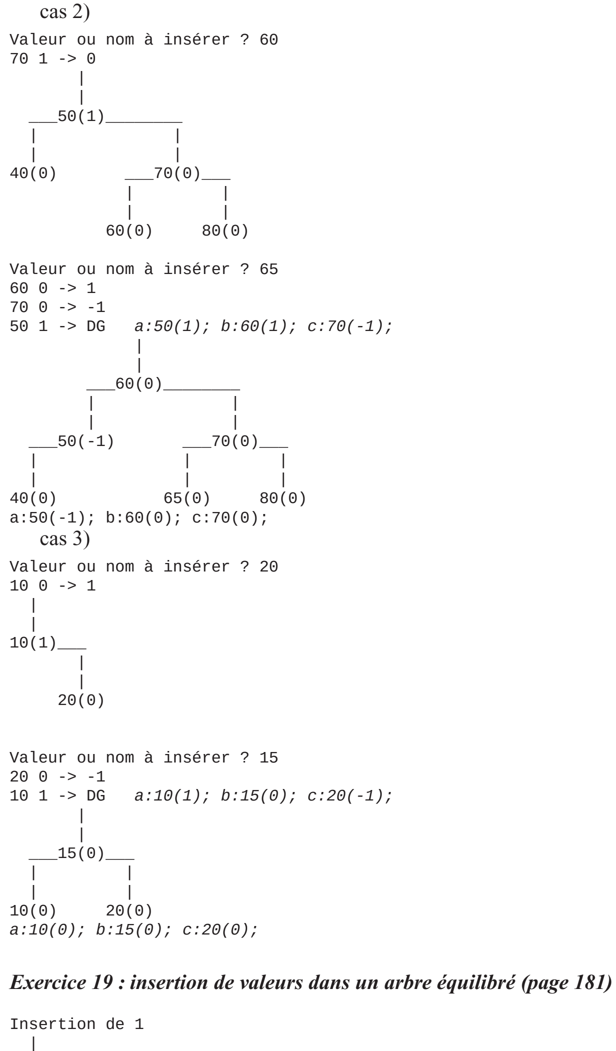


Exercice 18 : facteur d'équilibre (page 176)

cas 1)

```
Valeur ou nom à insérer ? 60
70 1 -> 0
       |
       |
  ___50(1)________
  |              |
  |              |
40(0)       ___70(0)___
            |         |
            |         |
          60(0)     80(0)


Valeur ou nom à insérer ? 58
60 0 -> -1
70 0 -> -1
50 1 -> DG   a:50(1); b:60(-1); c:70(-1);
                 |
                 |
     ________60(0)___
     |              |
     |              |
  ___50(0)___     70(1)___
  |         |            |
  |         |            |
40(0)     58(0)        80(0)
a:50(0); b:60(0); c:70(1);
```



cas 2)

```
Valeur ou nom à insérer ? 60
70 1 -> 0
       |
       |
  ___50(1)________
  |              |
  |              |
40(0)       ___70(0)___
            |         |
            |         |
          60(0)     80(0)

Valeur ou nom à insérer ? 65
60 0 -> 1
70 0 -> -1
50 1 -> DG   a:50(1); b:60(1); c:70(-1);
             |
             |
        ___60(0)________
        |              |
        |              |
  ___50(-1)       ___70(0)___
  |               |         |
  |               |         |
40(0)           65(0)     80(0)
a:50(-1); b:60(0); c:70(0);
```

cas 3)

```
Valeur ou nom à insérer ? 20
10 0 -> 1
  |
  |
10(1)___
       |
       |
     20(0)

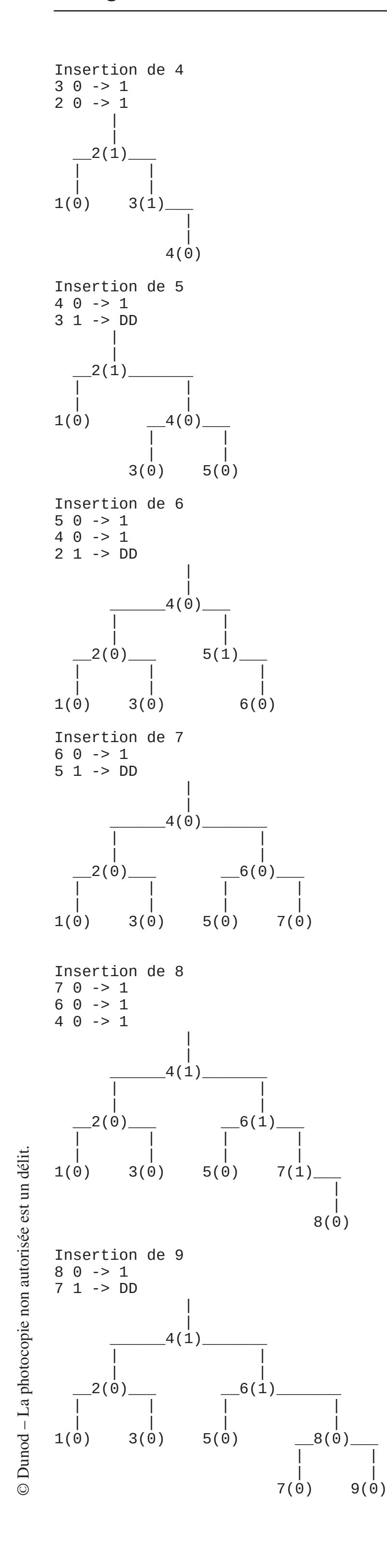

Valeur ou nom à insérer ? 15
20 0 -> -1
10 1 -> DG   a:10(1); b:15(0); c:20(-1);
       |
       |
  ___15(0)___
  |         |
  |         |
10(0)     20(0)
a:10(0); b:15(0); c:20(0);
```

Exercice 19 : insertion de valeurs dans un arbre équilibré (page 181)

```
Insertion de 1
  |
  |
1(0)

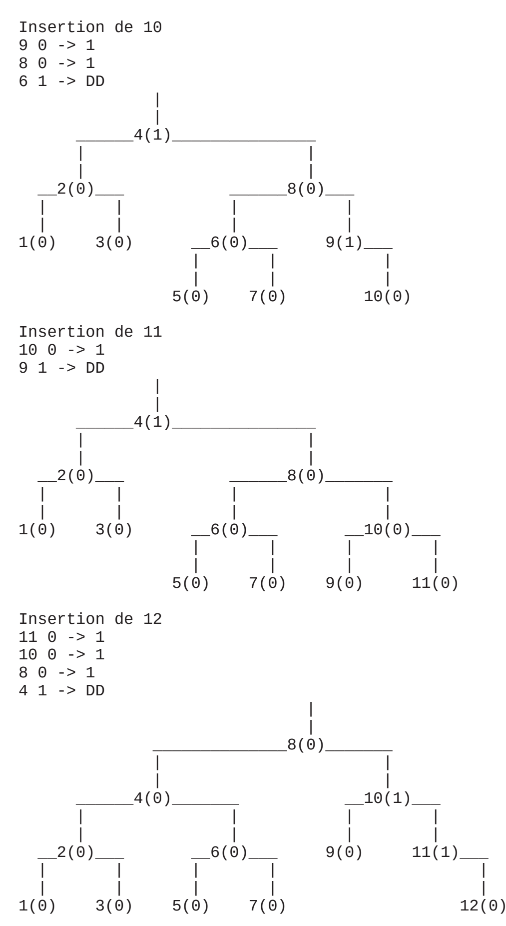
Insertion de 2
1 0 -> 1
  |
  |
1(1)___
      |
      |
    2(0)

Insertion de 3
2 0 -> 1
1 1 -> DD
      |
      |
  __2(0)___
  |       |
  |       |
1(0)    3(0)
```



```
Insertion de 4
3 0 -> 1
2 0 -> 1
      |
      |
  __2(1)___
  |       |
  |       |
1(0)    3(1)___
              |
              |
            4(0)
Insertion de 5
4 0 -> 1
3 1 -> DD
      |
      |
  __2(1)_______
  |           |
  |           |
1(0)      __4(0)___
          |       |
          |       |
        3(0)    5(0)
Insertion de 6
5 0 -> 1
4 0 -> 1
2 1 -> DD
              |
              |
      ______4(0)___
      |           |
      |           |
  __2(0)___     5(1)___
  |       |           |
  |       |           |
1(0)    3(0)        6(0)
Insertion de 7
6 0 -> 1
5 1 -> DD
              |
              |
      ______4(0)_______
      |               |
      |               |
  __2(0)___       __6(0)___
  |       |       |       |
  |       |       |       |
1(0)    3(0)    5(0)    7(0)

Insertion de 8
7 0 -> 1
6 0 -> 1
4 0 -> 1
              |
              |
      ______4(1)_______
      |               |
      |               |
  __2(0)___       __6(1)___
  |       |       |       |
  |       |       |       |
1(0)    3(0)    5(0)    7(1)___
                              |
                              |
                            8(0)
Insertion de 9
8 0 -> 1
7 1 -> DD
              |
              |
      ______4(1)_______
      |               |
      |               |
  __2(0)___       __6(1)_______
  |       |       |           |
  |       |       |           |
1(0)    3(0)    5(0)      __8(0)___
                          |       |
                          |       |
                        7(0)    9(0)
```



```
Insertion de 10
9 0 -> 1
8 0 -> 1
6 1 -> DD
              |
              |
      ______4(1)________________
      |                       |
      |                       |
  __2(0)___           ______8(0)___
  |       |           |           |
  |       |           |           |
1(0)    3(0)      __6(0)___     9(1)___
                  |       |           |
                  |       |           |
                5(0)    7(0)        10(0)

Insertion de 11
10 0 -> 1
9 1 -> DD
              |
              |
      ______4(1)________________
      |                       |
      |                       |
  __2(0)___           ______8(0)_______
  |       |           |               |
  |       |           |               |
1(0)    3(0)      __6(0)___       __10(0)___
                  |       |       |        |
                  |       |       |        |
                5(0)    7(0)    9(0)     11(0)

Insertion de 12
11 0 -> 1
10 0 -> 1
8 0 -> 1
4 1 -> DD
                              |
                              |
              ______________8(0)_______
              |                       |
              |                       |
      ______4(0)________          __10(1)___
      |               |           |        |
      |               |           |        |
  __2(0)___       __6(0)___     9(0)     11(1)___
  |       |       |       |                     |
  |       |       |       |                     |
1(0)    3(0)    5(0)    7(0)                  12(0)
```

Pour l'insertion des nombres de 12 à 1, on obtient des schémas symétriques de ceux donnés ci-dessus. L'insertion se fait toujours dans le sous-arbre gauche ; le rééquilibrage est donc un rééquilibrage GG au lieu de DD.

Exercice 20 : gestion d'un tournoi à l'aide d'une liste d'arbres (page 186)

```
/* tournoi.cpp  programme de gestion d'un tournoi
   méthode : utilisation d'une liste d'arbres;
   quand le tournoi est fini, il reste un seul arbre dans la liste */

#include <stdio.h>
#include <string.h>      // strcmp
#include <stdlib.h>      // malloc
#include "liste.h"
#include "arbre.h"

#define LGMAX 20
typedef char Chaine [LGMAX+1];  // 0 de fin
typedef struct {
  Chaine joueur1;    // gagnant = joueur1
  Chaine joueur2;
} Match;

// constructeur d'un match
Match* match (char* joueur1, char* joueur2) {
  Match* nouveau = new Match();
  strcpy (nouveau->joueur1, joueur1);
  strcpy (nouveau->joueur2, joueur2);
  return nouveau;
}
```

```
// la chaîne à fournir pour un match
char* toStringMatch (Match* match) {
  char* chaine = (char*) malloc (2*LGMAX+1);
  sprintf (chaine, "%s:%s", match->joueur1, match->joueur2);
  return chaine;
}

// égalité : 0; sinon 1
int comparerMatch (Match* match1, Match* match2) {
  // match2->joueur1 contient le joueur cherché
  // égalité si le joueur cherché match2->joueur1
  // est dans la structure match1 (en joueur1 ou joueur2)
  if (strcmp (match1->joueur1, match2->joueur1) == 0  ||
      strcmp (match1->joueur2, match2->joueur1) == 0 ) {
    return 0;
  } else {
    return 1;
  }
}

char* toStringMatch (Objet* objet) {
  return toStringMatch ( (Match*)objet);
}

int comparerMatch (Objet* objet1, Objet* objet2) {
  return comparerMatch ((Match*) objet1, (Match*)objet2);
}

// pour un match
void ecrireMatch (Match* match) {
  printf (" %s gagne contre %s\n", match->joueur1, match->joueur2);
}

// lister le contenu des racines des arbres de la liste,
// soit les joueurs non éliminés ayant déjà joués
void listerRestants (Liste* li) {
  printf ("\n\nJoueurs non éliminés\n");
  ouvrirListe (li);
  while (!finListe (li)) {
    Arbre* arbre = (Arbre*) objetCourant (li);
    Match* match  = (Match*) getobjet (getracine(arbre));
    ecrireMatch (match);
  }
  printf ("\n");
}

// lister le contenu de chaque arbre de la liste des arbres;
// récapitulatif des matchs
// type 1 : préfixée; 2 : postfixée; 3 : dessins des arbres
void listerArbres (Liste* li, int type) {
  ouvrirListe (li);
  while (!finListe (li) ) {
    Arbre* arbre = (Arbre*) objetCourant (li);
    switch (type) {
    case 1 :
      indentationPrefixee  (arbre, 1);
      break;
    case 2 :
      indentationPostfixee (arbre, 1);
      break;
    case 3 :
      printf ("\n");
      dessinerArbre (arbre, stdout);
      break;
    }
    printf ("\n");
  }
  printf ("\n");
}

// lister les matchs d'un joueur connaissant un pointeur sur
// son dernier match joué (le plus haut dans l'arbre)
void listerMatch (Noeud* pNom, Chaine joueur) {
  if (pNom != NULL) {
    Match* match = (Match*) getobjet(pNom);
    ecrireMatch (match);
    if (strcmp (match->joueur1, joueur) == 0) {
      listerMatch (getsag(pNom), joueur);
```

```
    } else {
      listerMatch (getsad(pNom), joueur);
    }
  }
}

// rechercher joueur dans la liste des arbres
// et lister les matchs du joueur
void listerMatch (Liste* li, Chaine joueur) {
  printf ("Matchs du joueur %s \n", joueur);
  booleen trouve   = faux;
  Match*  joueurCh = match (joueur, "");

  ouvrirListe (li);
  while (!finListe (li) && !trouve) {
    Arbre* arbre = (Arbre*) objetCourant (li);
    Noeud* noeud   = trouverNoeud (arbre, joueurCh);
    trouve = noeud != NULL;
    if (trouve) listerMatch (noeud, joueur);
  }
  if (!trouve) {
    printf ("Aucun match enregistré pour %s\n", joueur);
  }
}

// enregistrer le résultat d'un match dans la liste des arbres
void enregistrerMatch (Liste* li, Chaine j1, Chaine j2) {
  Noeud*  ptJ1 = NULL; // repère j1
  Noeud*  ptJ2 = NULL; // repère j2

  printf ("enregistrerMatch %s contre %s\n", j1, j2);
  ouvrirListe (li);

  // chercher le noeud ptj1 du dernier match de j1
  //        et le noeud ptj2 du dernier match de j2
  booleen fin = faux;
  while (!finListe (li) && !fin ) {
    Arbre* arbre = (Arbre*) objetCourant (li);
    Noeud* noeud = getracine (arbre);
    Match* match = (Match*) getobjet(noeud);
    if (strcmp (match->joueur1, j1) == 0) {
      ptJ1 = noeud;
      extraireUnObjet (li, arbre);
    }
    if (strcmp (match->joueur1, j2) == 0) {
      ptJ2 = noeud;
      extraireUnObjet (li, arbre);
    }
    fin = (ptJ1!=NULL) && (ptJ2!=NULL);
  }

  // créer un Match, un Noeud et un arbre
  Match* nvMatch = match (j1, j2);
  Noeud* nvNoeud = cNd (nvMatch, ptJ1, ptJ2);
  Arbre* arbre = creerArbre (nvNoeud, toStringMatch, comparerMatch);
  insererEnFinDeListe (li, arbre);
}

// créer la liste des arbres à partir d'un fichier
// des résultats des matchs joués
void creerArbres (FILE* fe, Liste* li) {
  initListe (li);
  while (!feof (fe)) {
    Chaine j1;
    Chaine j2;
    fscanf (fe, "%10s%10s", j1, j2);
    if (feof (fe)) break;
    //printf ("creerArbres j1 : %s, j2 : %s\n", j1, j2);
    enregistrerMatch (li, j1, j2);
  }
}


// Les différentes possibilités du programme
int menu () {
  printf ("\nGESTION D'UN TOURNOI\n\n");
  printf ("0 - Fin\n");
  printf ("1 - Création de la liste d'arbres à partir d'un fichier\n");
  printf ("2 - Enregistrement d'un match\n");
```

```
  printf ("3 - Liste des joueurs non éliminés\n");
  printf ("4 - Parcours préfixé  des arbres\n");
  printf ("5 - Parcours postfixé des arbres\n");
  printf ("6 - Dessins des arbres des matchs\n");
  printf ("7 - Liste des matchs d'un joueur\n");
  printf ("\n");

  printf ("Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  return cod;
}

void main () {
  Liste* la   = creerListe(); // la liste des arbres
  int    fini = faux;

  while (!fini) {

    switch (menu() ) {

    case 0:
      fini = vrai;
      break;

    case 1: {
      //printf ("Nom du fichier contenant les résultats des matchs ? ");
      char nomFE [50];
      //scanf  ("%s", nomFE);
      strcpy (nomFE, "jeu.dat");
      FILE* fe = fopen (nomFE, "r");
      if (fe == NULL) {
        printf ("%s inconnu\n", nomFE); exit (1);
      }
      creerArbres (fe, la);
      fclose (fe);
      } break;

    case 2 : {
      printf ("nom du gagnant ? ");
      Chaine j1; scanf ("%s", j1);
      printf ("nom du perdant ? ");
      Chaine j2; scanf ("%s", j2);
      enregistrerMatch (la, j1, j2);
      } break;

    case 3:
      listerRestants (la);
      break;

    case 4:
      printf ("préfixé indenté\n\n");
      listerArbres (la, 1);
      break;

    case 5:
      printf ("postfixé indenté\n\n");
      listerArbres (la, 2);
      break;

    case 6:
      printf ("dessins des arbres des matchs\n\n");
      listerArbres (la, 3);
      break;

    case 7: {
      printf ("\nMatchs d'un joueur, nom cherché ? ");
      Chaine joueur;      // nom du Joueur cherché
      scanf ("%s", joueur);
      listerMatch (la, joueur);
      } break;
    }  // switch

    if (!fini) {
      printf ("\n\nTaper Return pour continuer\n");
      getchar();
    }
  }  // while

} // main
```

Exercice 21 : mémorisation d'un dessin sous forme d'un arbre binaire (page 189)

```
/* dessin.cpp  Dessin mémorisé sous forme d'un arbre statique */

#include <stdio.h>
#include <stdlib.h>
#include "ecran.h"

#define NULLE -1
typedef int   PNoeud;

typedef struct {
  int    indice;   // indice sur desc []
  PNoeud gauche;
  PNoeud droite;
} Noeud;

int desc [] = {    // description de l'image
// 0                                            15
   3, 5, 5, 5, 3, 6, 6, 6, 7, 5, 5, 5, 5, 5, 5, 5,
   3, 7, 7, 7, 3, 3, 3, 3, 2, 5, 5
};

Noeud arbre [] = { // l'arbre représentant l'image
  { 0,   1, -1 },  // 0
  { 4,  -1,  2 },  // 1
  { 8,   3, -1 },  // 2
  { 16, -1,  4 },  // 3
  { 20, -1,  5 },  // 4
  { 24, -1, -1 }   // 5
};

// tracer le dessin correspondant à un noeud
void traiterNoeud (PNoeud pNd, int x, int y, int taille, int* nx, int* ny) {
  int ind = arbre[pNd].indice;
  int nb  = desc [ind];
  for (int i=ind+1; i<=ind+nb; i++) {
    for (int j=1; j<=taille; j++) {
      ecrirePixel (y, x);  // numéro de ligne d'abord
      switch (desc[i]) {
        case 1:           y = y-1; break;
        case 2: x = x+1;  y = y-1; break;
        case 3: x = x+1;           break;
        case 4: x = x+1;  y = y+1; break;
        case 5:           y = y+1; break;
        case 6: x = x-1;  y = y+1; break;
        case 7: x = x-1;           break;
        case 8: x = x-1;  y = y-1; break;
      }
    }
  }
  *nx = x;
  *ny = y;
}

// parcours et dessin de l'arbre
void parcours (PNoeud racine, int x, int y, int taille) {
  if (racine != NULLE) {
    int nx, ny;
    traiterNoeud (racine, x, y, taille, &nx, &ny);
    parcours     (arbre[racine].gauche, nx,  ny, taille);
    parcours     (arbre[racine].droite,  x,  y,  taille);
  }
}

// trace deux dessins : un grand et un petit
void main () {
  PNoeud racine = 0;  // premier noeud de l'arbre

  initialiserEcran (40, 40);
  parcours         (racine, 10, 10, 2);
  afficherEcran    ();
  sauverEcran      ("GrdEcran.res");
  detruireEcran    ();

  initialiserEcran (20, 20);
  parcours         (racine, 2, 2, 1);
  afficherEcran    ();
  sauverEcran      ("PtiEcran.res");
  detruireEcran    ();
}
```

Exercice 22 : références croisées (arbre ordonné) (page 192)

```
/* refcrois.cpp
   références croisées : listes et arbres */

#include <stdio.h>    // printf, FILE, fopen, fscanf
#include <string.h>   // strcpy, strcmp, strlen
#include <stdlib.h>   // exit
#include <ctype.h>    // isalpha
#include "liste.h"
#include "arbre.h"

// les objets Elt (liste de numéros de lignes)

typedef struct {
  int numLigne;
} Elt;

// les objets Mot
typedef char Chaine [31];  // 30 + 0 de fin
// un mot et sa liste de lignes
typedef struct {
  Chaine mot;
  Liste  li;
} Mot;

// LES FONCTIONS EN PARAMETRES

// utilisé par dessinerArbre() pour avoir les mots
char* toStringMot (Mot* pmot) {
  return pmot->mot;
}

// utilisé par dessinerArbre() pour avoir mots et numéros
char* toStringMot2 (Mot* pmot) {
  Liste* li = &pmot->li;
  ouvrirListe (li);
  char* reponse = (char*) malloc (50);
  char mot [10];
  sprintf (reponse, "%s:", pmot->mot);
  while (!finListe (li)) {
    Elt* nl = (Elt*) objetCourant (li);
    sprintf (mot, "%d ", nl->numLigne);
    strcat (reponse, mot);
  }
  return reponse;
}

// comparer deux Mot (pour rechercherOrd())
int comparerMot (Mot* mot1, Mot* mot2) {
  return strcmp (mot1->mot, mot2->mot);
}

// insérer mot, trouvé à la ligne nl, dans l'arbre ordonné
void insererMot (Arbre* arbre, Chaine mot, int nl) {
  Mot* motCherche = new Mot();
  strcpy (motCherche->mot, mot);
  // fournit un pointeur sur le noeud contenant motCherché
  // rechercherOrd() utilise comparerMot()
  Noeud* noeud = rechercherOrd (arbre, motCherche);

  if (noeud != NULL) {
    // le mot existe déjà; ajouter le numéro de ligne
    Mot* pmot = (Mot*) getobjet(noeud);
    Elt* elt = new Elt();
    elt->numLigne = nl;
    insererEnFinDeListe (&pmot->li, elt);

  } else {
    // première rencontre du mot : créer le Mot;
    // insérer nl en fin de liste (le premier élément)
    Mot* pmot = motCherche;
    initListe (&pmot->li);
    Elt* elt = new Elt();
    elt->numLigne = nl;
    insererEnFinDeListe (&pmot->li, elt);
    // insérer le Mot pointé par pmot dans l'arbre ordonné (ou équilibré)
    // en utilisant comparerMot pour trouver sa place
    insererArbreOrd (arbre, pmot);
    //insererArbreEquilibre (arbre, pmot);
  }
}
```

```
// parcours de la liste du Mot pointé par pmot;
// écriture de 10 valeurs par ligne
void traiterNd (Mot* pmot) {
  Liste* li = &pmot->li;
  printf ("%15s : ", pmot->mot);

  int i = 0;
  ouvrirListe (li);
  while (!finListe (li) ) {
    Elt* ptc = (Elt*) objetCourant (li);
    printf ("%6d", ptc->numLigne);
    i++;
    if ( (i % 10 == 0) && !finListe (li) ) {
      printf ("\n%18s", " ");
    }
  }
  printf ("\n");
}


// rechercher les numéros de ligne d'un mot
void rechercherLignes (Arbre* arbre, char* mot) {
  Mot* motCherche = new Mot();
  strcpy (motCherche->mot, mot);
  Noeud* noeud = rechercherOrd (arbre, motCherche);
  if (noeud != NULL) {
    traiterNd ((Mot*)getobjet(noeud));
  } else {
    printf ("%s inconnu\n", mot);
  }
}

// ce qui compose un mot : on pourrait y ajouter
// les lettres accentuées, les chiffres, etc.
int elementDeMot (char c) {
  return isalpha (c);
}

char* toStringMot (Objet* objet) {
  return toStringMot ((Mot*)objet);
}
char* toStringMot2 (Objet* objet) {
  return toStringMot2 ((Mot*)objet);
}

int comparerMot (Objet* objet1, Objet* objet2) {
  return comparerMot ((Mot*)objet1, (Mot*)objet2);
}

// pour infixe() dans main()
void traiterNd (Objet* objet) {
  traiterNd ( (Mot*) objet);
}

void main() {
  printf ("Nom du fichier dont on veut les références croisées ? ");
  char nomFE [50];        // nom du fichier d'entrée
  scanf ("%s", nomFE);
  //strcpy (nomFE, "refcrois.dat");
  FILE* fe = fopen (nomFE, "r");   // fichier d'entrée
  if (fe == NULL) {
    printf ("Erreur ouverture %s\n", nomFE);
    exit (1);
  }

  //Arbre* arbre = creerArbre (NULL, toStringMot, comparerMot);
  Arbre* arbre = creerArbre (NULL, toStringMot2, comparerMot);

  int    nl     = 1;      // numéro de ligne du fichier
  printf ("%6d ", nl);

  char    c;              // prochain caractère à traiter (analyseur)
  fscanf (fe, "%c", &c);  printf ("%c", c);
  Chaine  mot;            // mémorise le mot lu dans fe

  while (! feof (fe)) {

    // passer les délimiteurs de mots
```

```
    while ( !feof (fe) && !elementDeMot (c) ) {
      if ( c == '\n') {
        nl = nl + 1;
        printf ("%6d ", nl);
      }
      fscanf (fe, "%c", &c);  printf ("%c", c);
    }

    // lire les caractères du mot
    char* pMot = mot;    // pointeur courant sur Mot
    while (!feof(fe) && elementDeMot (c) ) {
      *pMot++ = c;
      fscanf (fe, "%c", &c);  printf ("%c", c);
    }
    *pMot = 0;
    //printf ("mot : %s\n", mot);

    if (strlen (mot) > 0)  {
      insererMot (arbre, mot, nl);
    }
  }
  fclose (fe);

  printf ("\n\nRéférences croisées\n");
  infixe (arbre, traiterNd);

  printf ("\n\nRéférences croisées de :\n");
  rechercherLignes (arbre, "petit");

  dessinerArbre (arbre, stdout);
}  // main
```

Exercice 23 : arbre de questions (page 194)

```
/* question.cpp  arbre de questions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>       // toupper
#include "arbre.h"
#include "mdtypes.h"     // toChar

const int MAX = 100;

void insererQuestion (Noeud** pracine) {
  Noeud* racine = *pracine;

  if (racine==NULL) {
    printf ("Question ? : ");

    char phrase [MAX]; fgets (phrase, MAX, stdin);
    phrase [strlen(phrase)-1] = 0;  // enlever \n
    char* question = (char*) malloc (strlen(phrase)+1);
    strcpy (question, phrase);
    Noeud* nouveau = cNd (question);
    *pracine = nouveau;
  } else {
    printf ("%s (O/N) ? ", (char*) getobjet(racine));
    char rep; scanf ("%c", &rep); getchar();
    if (toupper(rep) == 'O' ) {
      insererQuestion (&racine->gauche);
    } else {
      insererQuestion (&racine->droite);
    }
  }
}

void insererQuestion (Arbre* arbre) {
  insererQuestion (&arbre->racine);
}

void poserQuestions (Noeud* racine) {
  if (racine != NULL) {
    printf ("%s (O/N) ? ", (char*) getobjet(racine));
    char rep; scanf ("%c", &rep); getchar();
    if (toupper(rep) == 'O' ) {
      if (racine->gauche == NULL) {
        printf ("Fin des questions : vous avez trouvé\n");
```

```
      } else {
        poserQuestions (racine->gauche);
      }
    } else {
      if (racine->droite == NULL) {
        //printf ("Mystère et boule de gomme!\n");
        printf ("Fin des questions : ");
        printf ("je donne ma langue au chat\n");
      } else {
        poserQuestions (racine->droite);
      }
    }
  }
}

void poserQuestions (Arbre* arbre) {
  poserQuestions (getracine(arbre));
}

void sauverQuestions (Noeud* racine, FILE* fs) {
  if (racine==NULL) {
    fprintf (fs, "*\n");
  } else {
    fprintf (fs, "%s\n", (char*) getobjet(racine));
    sauverQuestions (racine->gauche, fs);
    sauverQuestions (racine->droite, fs);
  }
}

void sauverQuestions (Arbre* arbre, FILE* fs) {
  sauverQuestions (getracine(arbre), fs);
}

void chargerQuestions (Noeud** pracine, FILE* fe) {
  char phrase [MAX];
  fgets (phrase, MAX, fe);
  int lg = strlen (phrase);
  phrase [lg-1] = 0;

  if (phrase[0] == '*') {
    *pracine = NULL;
  } else {
    char* reference = (char*) malloc (strlen(phrase)+1);
    strcpy (reference, phrase);
    Noeud* nouveau = cNd (reference);
    *pracine = nouveau;

    chargerQuestions (&nouveau->gauche, fe);
    chargerQuestions (&nouveau->droite, fe);
  }
}

void chargerQuestions (Arbre* arbre, FILE* fe) {
  chargerQuestions (&arbre->racine, fe);
}

// détruire un objet de type Question
void detruireQuestion (Objet* objet) {
  char* message = (char*) objet;
  free (message);
}

int menu () {
  printf ("\nARBRE DE QUESTIONS\n\n");
  printf (" 0 - Fin\n");
  printf (" 1 - Insérer  une nouvelle question\n");
  printf (" 2 - Lister   l'arbre des questions\n");
  printf (" 3 - Poser    les questions\n");
  printf (" 4 - Sauver   l'arbre dans un fichier\n");
  printf (" 5 - Charger  l'arbre à partir d'un fichier\n");
  printf (" 6 - Dessiner l'arbre des questions\n");
  printf (" 7 - Détruire l'arbre en mémoire\n");
  printf ("\n");

  printf (" Votre choix ? ");
  int cod; scanf ("%d", &cod); getchar();
  return cod;
}

void main () {
  Arbre* arbre = creerArbre();
  booleen fini = faux;
```

```
  while (!fini) {
    switch ( menu() ) {
    case 0:
      fini = vrai;
      break;
    case 1 :
      printf ("Insérer une question\n");
      insererQuestion (arbre);
      break;
    case 2 :
      printf ("Arbre des questions\n");
      indentationPrefixee (arbre, 1);
      printf ("\n");
      break;
    case 3 :
      printf ("Questions\n");
      poserQuestions (arbre);
      break;
    case 4 : {
      printf ("Sauvegarde des questions\n");
      char nomFS [50];
      //printf ("Nom du fichier à créer ? ");
      //fgets (nomFS, 50, stdin);
      strcpy (nomFS, "question.dat");
      FILE* fs = fopen (nomFS, "w");
      sauverQuestions (arbre, fs);
      fclose (fs);
      printf ("Sauvegarde effectuée\n");
      } break;
    case 5 : {
      printf ("Chargement des questions\n");
      char nomFE [50];
      //printf ("Nom du fichier à charger ? ");
      //fgets (nomFE, 50, stdin);
      strcpy (nomFE, "question.dat");
      FILE* fe = fopen (nomFE, "r");
      chargerQuestions (arbre, fe);
      fclose (fe);
      printf ("Chargement effectué\n");
      dessinerArbre (arbre, stdout);
      } break;
    case 6 :
      dessinerArbre (arbre, stdout);
      break;
    case 7 :
      detruireArbre (arbre, detruireQuestion);
      printf ("Arbre détruit");
      break;
    }  // switch

    if (!fini) {
      printf ("Return pour continuer\n\n"); getchar();
    }
  }  // while
}
```

Exercice 24 : menu pour une table de hachage (page 237)

```
/* pphc.cpp  programme principal hashcode
   avec ou sans chaînage suivant la variable
   de compilation CHAINAGE
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "mdtypes.h"

#ifndef CHAINAGE
#include "tablehc.h"
typedef TableHC Table;
#else
#include "tablehcc.h"
typedef TableHCC Table;
#endif
```

```
// Les fonctions de hash-code
char* titreFn [] = {
  "somme des rangs alphabétiques",
  "somme des caractères ascii",
  "division par N",
  "changement de base"
};
#define NBFN 4

// Les résolutions
char* titreRe [] = {
  "r(i) = i",
  "r(i) = K*i",
  "r(i) = i*i",
  "pseudo-aléatoire",
};
#define NBRE 4

Table* creerTable (int nMax, char* (*toString) (Objet*),
                            int (*comparer) (Objet*, Objet*),
                            int (*hashcode) (Objet*, int),
                            int (*resolution) (int, int, int)) {
  #ifndef CHAINAGE
  return creerTableHC  (nMax, toString, comparer, hashcode, resolution);
  #else
  return creerTableHCC (nMax, toString, comparer, hashcode, resolution);
  #endif
}

int menu () {
  #ifndef CHAINAGE
  printf ("\n\nTABLE (HASH-CODING)\n\n");
  #else
  printf ("\n\nTABLE (HASH-CODING AVEC CHAINAGE)\n\n");
  #endif
  printf ( " 0 - Fin\n");
  printf ( " 1 - Initialisation de la table\n");
  printf ( " 2 - Hash-code d'un élément\n");
  printf ( " 3 - Ordre de test des N-1 entrées\n");
  printf ( " 4 - Ajout d'un élément dans la table\n");
  printf ( " 5 - Ajout d'éléments à partir d'un fichier\n");
  printf ( " 6 - Liste de la table\n");
  printf ( " 7 - Recherche d'une clé\n");
  printf ( " 8 - Collisions à partir d'une entrée\n");
  printf ("\n");
  printf ("Votre choix ? ");

  int cod; scanf ("%d", &cod); getchar();
  printf ("\n");

  return cod;
}

void lireFichier (char* nomFE, Table* table) {
  FILE* fe = fopen (nomFE, "r");
  if (fe==NULL) { perror (nomFE); exit(1); };
  int c;

  while (!feof (fe) ) {
    Personne* nouveau = new Personne();
    // lire la clé
    char* pNom  = nouveau->nom;
    while ( (c=getc(fe)) != ' ') *pNom++ = c;
    *pNom = 0;
    if (feof(fe)) break;
    // passer les espaces
    while ((c=getc(fe)) == ' ');
    // lire les infos
    pNom    = nouveau->prenom;
    *pNom++ = c;
    while ( ((c=getc(fe)) != '\n') && !feof (fe) ) {
      *pNom++ = c;
    }
    *pNom = 0;
    insererDsTable (table, nouveau);
  }
}
```

```
void main () {

  Table* table = creerTable (16, toStringPersonne, comparerPersonne, hash1, resolution1);

  booleen  fin  = faux;
  while (!fin) {

    switch (menu()) {
    case 0:
      fin = vrai;
      break;

    case 1: {
      printf ("Paramètres\n");
      printf ("Longueur N de la table ? ");
      int n; scanf  ("%d", &n);
      printf ("Fonctions de hash-code\n");
      for (int i=1; i<=NBFN; i++) printf ("%3d %s\n", i, titreFn[i-1]);
      printf ("Votre choix ? ");
      int typFn; scanf  ("%d", &typFn);
      if (typFn<1 || typFn>NBFN) typFn = 1;
      printf ("Résolution\n");
      for (int i=1; i<=NBRE; i++) printf ("%3d %s\n", i, titreRe[i-1]);
      printf ("Votre choix ? ");
      int typRes; scanf  ("%d", &typRes);
      if (typRes<1 || typRes>NBRE) typRes = 1;
      int (*hash) (Objet*, int);
      switch (typFn) {
      case 1 : hash = hash1; break;
      case 2 : hash = hash2; break;
      case 3 : hash = hash3; break;
      case 4 : hash = hash4; break;
      }
      int (*resolution) (int, int, int);
      switch (typRes) {
      case 1 : resolution = resolution1; break;
      case 2 : resolution = resolution2; break;
      case 3 : resolution = resolution3; break;
      case 4 : resolution = resolution4; break;
      }
      table = creerTable (n, toStringPersonne, comparerPersonne, hash, resolution);
      } break;

    case 2:{
      printf ("Clé dont on veut le hash-code ? ");
      char cle[16]; scanf  ("%s", cle);
      int h = table->hashcode (cle, table->nMax);
      printf ("Hash-Code : %d\n", h);
      } break;

    case 3: {
      printf ("Entrée du Hash-Code de départ ? ");
      int entree; scanf  ("%d", &entree);
      ordreResolution (table, entree);
      } break;

    case 4 : {
      Personne* p = creerPersonne();
      insererDsTable (table, p);
      } break;

    case 5 : {
      printf ("Nom du fichier ? ");
      char nomFE [30]; scanf ("%s", nomFE);
      lireFichier (nomFE, table);
      } break;

    case 6 :
      listerTable (table);
      break;

    case 7 : {
      printf ("nom de la clé de l'objet cherché ? ");
      char nom[20]; scanf ("%s", nom); getchar();
      Personne* cherche = creerPersonne (nom, "?");
      Personne* trouve = (Personne*) rechercherTable (table, cherche);
      if (trouve!= NULL) printf ("trouve %s\n", toStringPersonne (trouve));
      } break;
```

```
    case 8 : {
      printf ("Liste des collisions pour l'entrée ? ");
      int entree; scanf  ("%d", &entree);
      listerEntree (table, entree);
      } break;
    }
  } // while
}  // main
```

Exercice 25 : hachage avec chaînage dans une seule table (page 244)

```
booleen insererDsTable (TableHCC* table, Objet* nouveau) {
  int h = table->hashcode (nouveau, table->nMax);
  //printf ("insererDsTable hashcode %3d pour %s\n", h, table->toString(nouveau));
  if (table->element[h].objet == NULL) {
    // l'entrée est libre, on l'occupe
    //printf ("insererDsTable h2 %d\n", h);
    table->element[h].objet = nouveau;
    table->n++;
  } else {
    // entrée occupée
    Objet* occupant = table->element[h].objet;
    int hcOc = table->hashcode (occupant, table->nMax);
    if (hcOc == h) { // synonyme, on insère en tête de liste
      int re = resolution (table, h);
      if (re != -1) {
        // insertion de nouveau en tête de liste
        table->element[re] = table->element[h];
        table->element[h].objet   = nouveau;
        table->element[h].suivant = re;
        table->n++;
      } else {
        printf ("Table saturée\n");
        return faux;
      }
    } else {
      // enlever l'occupant de sa liste commençant en hcOc
      // ne pas augmenter table->n
      int i = hcOc;
      int prec;
      while (i != h) {
        prec = i;
        i    = table->element[i].suivant;
      }
      table->element[prec].suivant = table->element[h].suivant;

      // insérer nouveau en h
      table->element[h].objet   = nouveau;
      table->element[h].suivant = NILE;

      // réinsérer l'expulsé
      insererDsTable (table, occupant);
    }
  }
  return vrai;
}

// lister la table
void listerTable (TableHCC* table) {
  int sn = 0;
  for (int i=0; i<table->nMax; i++) {
    if (table->element[i].objet != NULL) {
      int n = nbAcces (table, table->element[i].objet);
      printf ("%3d : hc:%3d n:%3d svt:%3d %s\n", i,
                table->hashcode (table->element[i].objet, table->nMax), n,
                                                table->element[i].suivant,
                            table->toString (table->element[i].objet));
      if (n>0) sn += n;      //-1 si erreur
    } else {
      printf ("%3d :\n", i);
    }
  }

  printf ("\nNombre d'éléments dans la table : %d", table->n);
  printf ("\nTaux d'occupation de   la table : %.2f",
      table->n / (double) table->nMax);
  printf ("\nNombre moyen d'accès à la table : %.2f\n\n",
      sn / (double) table->n);
}
```

```
// fournir le nombre d'accès à la table
// pour retrouver objetCherche; -1 si inconnu
int nbAcces (TableHCC* table, Objet* objetCherche) {
  int na = 0;                  // nombre d'accès
  int hc = table->hashcode (objetCherche, table->nMax);  // hash-code
  if (table->element[hc].objet == NULL) {
    na = -1;                            // élément inconnu
  } else {
    int courant = hc;
    int i       = 1;                    // ième tentative
    na++;
    while ( table->comparer (objetCherche, table->element[courant].objet) != 0) {
      courant = table->element[courant].suivant;
      if (courant == -1) return -1;    // élément inconnu
      na++;
    }
  }
  return na;
}
```

Exercice 26 : hachage sur l'arbre de la Terre (page 252)

```
   2  1             Bretagne  -1  60  -1
   7  1              Afrique  53  19  -1
   8  1             Belgique  -1  70  -1
  10  1               Europe  47  34  -1
  19  1             Amerique  -1  52  -1
  32  1                 Inde  -1  72  -1
  34  1                 Asie  39   7  71
  39  1                Chine  -1  32  72
  47  1               France   2  67  -1
  52  1              Oceanie  -1  -1  -1
  53  1                Niger  -1  54  -1
  54  1                Congo  -1  -1  -1
  56  1                Japon  -1  -1  -1
  60  1                Corse  -1  71  -1
  66  1                Terre  10  -1  -1
  67  1              Espagne  -1   8  70
  70  1             Danemark  -1  -1  -1
  71  1            Bourgogne  -1  -1  -1
  72  1                 Irak  -1  56  -1
```

Exercice 27 : table des étudiants gérée par hachage avec chaînage (page 252)

```
/* pphcetud.h  programme principal hashcode etudiant
               avec ou sans chaînage */

#include <stdio.h>
#include <stdlib.h>    // exit
#include "mdtypes.h"   // Personne

#ifndef CHAINAGE
#include "tablehc.h"
typedef TableHC Table;
#else
#include "tablehcc.h"
typedef TableHCC Table;
#endif

// résolution r(i) = k*i
int resolution22 (int h, int n, int i) {
  int k = 19;
  return (h+k*i) % n;
}

void lireFichier (char* nomFE, Table* table)                              voir exercice 24

void main() {
  char* nomFE = "etudiant.dat";

  #ifndef CHAINAGE
  TableHC*  table = creerTableHC  (197, toStringPersonne,
                                     comparerPersonne, hash2, resolution22);
  #else
  TableHCC* table = creerTableHCC (197, toStringPersonne,
                                     comparerPersonne, hash2, resolution22);
```

```
  #endif

  lireFichier (nomFE, table);
  //printf ("nombre moyen d'accès : %.2f\n", nbMoyAcces (table));
  listerTable (table);

  Personne* etu1 = creerPersonne ("TASSEL", "?");
  Personne* trouve = (Personne*) rechercherTable (table, etu1);
  printf ("%s\n", table->toString (trouve));
}
```

Exemple de table obtenue à partir d'un fichier « etudiant.dat » ordonné suivant le nom des étudiants.

```
  0 : hc:  0 n:  1 svt: -1 VERON Pascal
  1 : hc:  1 n:  1 svt: -1 CRAMBERT Pascal
  2 :
  3 :
  4 :
  5 : hc:145 n:  2 svt: -1 LE_NEDELEC Laurent
  6 :
  7 : hc:  7 n:  1 svt: 26 LOUSSOUARN Johann
  8 : hc:  8 n:  1 svt: 27 SOYER Benoît
  9 :
 10 : hc: 10 n:  1 svt: -1 BOURDAIS Candylène
 11 : hc: 11 n:  1 svt: -1 GUIRRIEC Stéphane
 12 :
 13 : hc:172 n:  2 svt: -1 DREAU Sylvain
 14 : hc: 14 n:  1 svt: -1 GUYOT Eric
 15 :
 16 : hc: 16 n:  1 svt: -1 LEVALOIS Fabien
 17 :
 18 : hc: 18 n:  1 svt: -1 LE_GLOAN Guenhaël
 19 :
 20 : hc: 20 n:  1 svt: 39 LE_LOCAT Yann-Gaël
 21 :
 22 :
 23 :
 24 : hc: 24 n:  1 svt:119 LOURENCO Gabriel
 25 :
 26 : hc:  7 n:  2 svt: -1 KERDRAON Benoît
 27 : hc:  8 n:  2 svt: -1 LARRONDE Stéphane
 28 :
 29 :
 30 :
 31 : hc: 31 n:  1 svt: -1 MER Patricia
 32 : hc: 32 n:  1 svt: -1 CAUDAL Vonig
 33 :
 34 :
 35 : hc: 35 n:  1 svt: 54 LE_MOUEL Jérôme
 36 :
 37 :
 38 :
 39 : hc: 20 n:  2 svt: -1 JAN Grégory
 40 :
 41 :
 42 : hc: 42 n:  1 svt: -1 BLASCO Nathalie
 43 : hc: 43 n:  1 svt:157 RONDEPIERRE Laurent
 44 :
 45 :
 46 : hc: 46 n:  1 svt: -1 GAUDIN Damien
 47 : hc: 47 n:  1 svt: -1 LE_GUILCHER Anne Claire
 48 :
 49 :
 50 :
 51 : hc: 77 n:  2 svt: -1 KERMARREC Patrice
 52 : hc: 52 n:  1 svt: -1 MONTEMBAULT Sébastien
 53 : hc: 53 n:  1 svt: -1 ROY Stéphane
 54 : hc: 35 n:  2 svt: -1 DANIEL Gilles
 55 :
 56 :
 57 :
 58 : hc: 58 n:  1 svt: -1 SALAUN Olivier
 59 :
 60 : hc: 60 n:  1 svt: -1 ORIERE Isabelle
 61 : hc: 61 n:  1 svt: 80 PIERRE Patrick
 62 : hc: 62 n:  1 svt:176 SEZNEC Olivier
 63 :
 64 : hc: 64 n:  1 svt: -1 MEULIN Benoît
 65 :
 66 : hc: 66 n:  1 svt: 85 TASSEL Jérôme
```

```
 67 :
 68 : hc: 68 n:  1 svt: -1 ROBERT Sébastien
 69 :
 70 :
 71 :
 72 :
 73 : hc: 73 n:  1 svt:168 LE_HENAFF Guénaëlle
 74 : hc: 74 n:  1 svt: -1 LE_POITEVIN Bertrand
 75 : hc: 75 n:  1 svt:170 REBOUX Franck
 76 : hc: 76 n:  1 svt: -1 LE_LAY Stéphane
 77 : hc: 77 n:  1 svt: 51 LE_GALLIC Régis
 78 :
 79 :
 80 : hc: 61 n:  2 svt: -1 PERIER Fabrice
 81 : hc: 81 n:  1 svt: -1 COUTELLEC Christophe
 82 : hc: 82 n:  1 svt: -1 UNVOAS Thierry
 83 :
 84 :
 85 : hc: 66 n:  2 svt: -1 GUILLO Yvan
 86 : hc: 86 n:  1 svt: -1 DEMEURANT Anne
 87 :
 88 :
 89 : hc: 89 n:  1 svt:108 SEYNHAEVE Emmanuel
 90 : hc: 90 n:  1 svt: -1 JACQ Armelle
 91 :
 92 : hc: 92 n:  1 svt: -1 HENNEQUIN Ludovic
 93 : hc: 93 n:  1 svt: -1 QUEMERAIS Claude
 94 : hc: 94 n:  1 svt: -1 KOWTUN Régis
 95 :
 96 : hc: 96 n:  1 svt: -1 LE_ROY Arnaud
 97 :
 98 :
 99 : hc: 99 n:  1 svt:118 LAIR Olivier
100 : hc: 62 n:  3 svt: -1 BRUNEL Pierre-Yves
101 :
102 : hc:102 n:  1 svt:121 VITTOZ Cécile
103 :
104 :
105 :
106 : hc:106 n:  1 svt: -1 LEYE Daouda
107 : hc:107 n:  1 svt: -1 CARREGA Philippe
108 : hc: 89 n:  2 svt: -1 LEMETAYER Valérie
109 :
110 :
111 : hc: 73 n:  3 svt: -1 DESCHAMPS Loic
112 :
113 : hc: 75 n:  3 svt: -1 BOUVET Yann
114 : hc:114 n:  1 svt:133 PRAT Yann
115 : hc:115 n:  1 svt: -1 RICHARD Didier
116 : hc:116 n:  1 svt: -1 BESCOND Arnaud
117 : hc:117 n:  1 svt: -1 DINCUFF Thierry
118 : hc: 99 n:  2 svt: -1 L'ESCOP Katia
119 : hc: 24 n:  2 svt: -1 MAO Véronique
120 :
121 : hc:102 n:  2 svt: -1 LE_CALVEZ Davy
122 :
123 :
124 :
125 : hc:125 n:  1 svt: -1 HERLIDO Jérôme
126 : hc:126 n:  1 svt: -1 ETIENNE Sébastien
127 : hc:127 n:  1 svt: -1 NICOLAS Christelle
128 : hc:128 n:  1 svt: -1 CLOATRE Gildas
129 :
130 : hc:130 n:  1 svt: -1 SALUDEN Yann
131 :
132 : hc:132 n:  1 svt:151 KERRIEL Philippe
133 : hc:114 n:  2 svt: -1 DELHAYE Sébastien
134 : hc:134 n:  1 svt:191 LESAINT Nicolas
135 : hc:135 n:  1 svt: -1 BOISSEL Jean-Christophe
136 : hc:136 n:  1 svt: -1 FOSSARD Arnaud
137 : hc:137 n:  1 svt:175 TORCHEN Yannick
138 : hc:138 n:  1 svt: -1 QUEMERE Marc
139 :
140 :
141 :
142 :
143 :
144 :
145 : hc:145 n:  1 svt:  5 RENAULT David
```

```
146 :
147 :
148 : hc:148 n:  1 svt: -1 LE_MENN Erwann
149 : hc:149 n:  1 svt: -1 LE_GUEN Grégory
150 :
151 : hc:132 n:  2 svt: -1 AUFFRAY céline
152 : hc:152 n:  1 svt: -1 LE_MAOU Yann
153 : hc:134 n:  3 svt: -1 COSTARD Sylvain
154 : hc:154 n:  1 svt: -1 QUENTIN Thierry
155 :
156 : hc:137 n:  3 svt: -1 FOURBIL Stéphane
157 : hc: 43 n:  2 svt: -1 MOALIC Denis
158 : hc:158 n:  1 svt: -1 CABON Ronan
159 :
160 :
161 :
162 : hc:162 n:  1 svt: -1 CHEIN Anthony
163 :
164 : hc:164 n:  1 svt: -1 HERVAGAULT Stéphane
165 :
166 :
167 : hc:167 n:  1 svt:186 LOUVOIS Prébagarane
168 : hc: 73 n:  2 svt:111 MORVAN Alain
169 :
170 : hc: 75 n:  2 svt:113 AUTRET Frédéric
171 :
172 : hc:172 n:  1 svt: 13 RAMEL Olivier
173 : hc:173 n:  1 svt: -1 BETIN Béatrice
174 :
175 : hc:137 n:  2 svt:156 PORHIEL Pierrick
176 : hc: 62 n:  2 svt:100 BERESCHEL Frédéric
177 :
178 :
179 : hc:179 n:  1 svt: -1 URBAN Frédéric
180 : hc:180 n:  1 svt: -1 LE_DALLOUR David
181 : hc:181 n:  1 svt: -1 LEVEN David
182 : hc:182 n:  1 svt: -1 LE_BELLOUR Marina
183 : hc:183 n:  1 svt: -1 BALANANT Ronan
184 :
185 :
186 : hc:167 n:  2 svt: -1 FLOCH Mickaël
187 :
188 :
189 : hc:189 n:  1 svt: -1 LE_BOURHIS Thierry
190 : hc:190 n:  1 svt: -1 BOUCHARD David
191 : hc:134 n:  2 svt:153 LE_GALL Erwan
192 :
193 : hc:193 n:  1 svt: -1 LANGLAIS Yann
194 :
195 : hc:195 n:  1 svt: -1 TSEMO Edith
196 :

Nombre d'éléments dans la table : 107
Taux d'occupation de   la table : 0.54
Nombre moyen d'accès à la table : 1.30
```

Exercice 28 : fermeture transitive par somme de produits de matrices (page 293)

```
static void ecrireFMEtape (Matrice pn, int l, int ne, int nMax) {

  if (l==-1) {
    printf ("Nombre de chemins de longueur <= %d\n", ne);
  } else {
    printf ("Nombre de chemins de longueur = %d\n", l);
  }
  for (int i=0; i<ne; i++) {
    for (int j=0; j<ne; j++) printf ("%3d", pn [i*nMax+j]);
    printf ("%10s", " ");
    for (int j=0; j<ne; j++) {
      printf ("%3s", pn [i*nMax+j]!=0 ? "V" : "F");
    }
    printf ("\n");
  }
  printf ("\n");
}
```

```
// affectation de matrices : mc = ms
static void affMat (Matrice mc, Matrice ms, int ne, int nMax) {
 for (int i=0; i<ne; i++) {
   for (int j=0; j<ne; j++) {
     mc [i*nMax+j] = ms [i*nMax+j];
   }
 }
}

// cumul de matrices : mc = mc + ms
static void addMat (Matrice mc, Matrice ms, int ne, int nMax) {
 for (int i=0; i<ne; i++) {
   for (int j=0; j<ne; j++) {
     mc [i*nMax+j] += ms [i*nMax+j];
   }
 }
}

// produit de matrices : pn = d * m
static void prodMat (Matrice pn, Matrice d, Matrice m, int ne, int nMax) {
  for (int i=0; i<ne; i++) {
    for (int j=0; j<ne; j++) {
      pn [i*nMax+j] = 0;
      for (int k=0; k<ne; k++) {
        pn [i*nMax+j] += d [i*nMax+k] * m [k*nMax+j] ;
      }
    }
  }
}

static void ecrireMat (Matrice m, int ne, int nMax) {
  for (int i=0; i<ne; i++) {
    for (int j=0; j<ne; j++) {
      printf ("%4d ", m [i*nMax+j]);
    }
    printf ("\n");
  }
  printf ("\n");
}

void produitEtFermeture (GrapheMat* graphe) {
  int nMax = graphe->nMax;
  int ns   = graphe->n;

  Matrice pn = (int*) malloc (sizeof(int)*nMax*nMax);
  Matrice sm = (int*) malloc (sizeof(int)*nMax*nMax);
  Matrice d  = (int*) malloc (sizeof(int)*nMax*nMax);

  affMat (d,  graphe->element, ns, nMax);
  affMat (sm, graphe->element, ns, nMax);

  ecrireFMEtape (d, 1, ns, nMax);

  for (int i=2; i<=ns; i++) {
    prodMat       (pn, d,  graphe->element, ns, nMax) ;
    addMat        (sm, pn, ns, nMax);
    affMat        (d,  pn, ns, nMax);
    ecrireFMEtape (pn, i,  ns, nMax);
  }

  ecrireFMEtape (sm, -1, ns, nMax);

  free (pn);
  free (sm);
  free (d);
}
```

Bibliographie

Livres en français

A.Aho, J. Hopcraft, J. Ullman, *Structures de données et algorithmes*, InterEditions, 1987.
(Types de données abstraites, arbres, ensembles, graphes, tris, algorithmes, gestion de la mémoire.)

C. Carrez, *Structures de données en Java, C++ et Ada 95*InterEditions, 1997.
(Introduction, les structures de base, algorithmes de tri, la recherche, problèmes et solutions.)

J. Courtin, I. Kowarski, *Initiation à l'algorithmique et aux structures de données*, volume 2 – listes linéaires chaînées, structures linéaires particulières, les tables, les arbres, Dunod 1995

M. Divay, *Java et la programmation objet*, Dunod, 2002, ISBN 2 10 005891 6.

M. Divay, *Unix, Linux et les systèmes d'exploitation*, Dunod 2004, ISBN 2 10 007451 2.

C. Froidevaux, M.C. Gaudel, M. Soria, *Types de données et algorithmes*, McGraw-Hill.
(Introduction à l'analyse des algorithmes, structures de données, algorithmes de recherche, algorithme de tri, quelques algorithmes sur les graphes.)

J. Guyot, C. Vial, *Arbres, Tables et Algorithmiques*, Eyrolles, 1988.
(Algorithmique, structures de données, algorithmes.)

E. Horowitz, S. Sahni, S. Anderson-Freed, *L'essentiel des structures de données en C*, Dunod 1993.
(Concepts de base, piles et files, les listes, les arbres, les graphes, le tri, le hachage, les structures de tas, les structures arborescentes.)

B. Ibrahim, C. Pellegrini, *Structuration des données informatiques,* Dunod, 1989.
(Structures statiques et dynamiques, chaînes, arbres, listes, graphes, réseaux, adressage associatif, arbres de décision.)

Niklaus Wirth, *Algorithmes et structures de données*, Eyrolles, 1987.
(Structures de données fondamentales, le tri, algorithmes récursifs, structures de données dynamiques, transformations de clés (hachage).)

Livres en anglais

A. Tenenbaum, M. Augenstein, *Data structures using Pascal*, Prentice-Hall, 1981.
(Information and meaning, the stack, recursion, queues and lists, Pascal list processing, trees, graphs and their applications, sorting, searching.)

R. Johnsonbaugh, *Discrete mathematics*, Macmillan Publishing Company, 1984.
(Introduction, counting methods and recurrence relations, graph theory, trees, network models and Petri nets, boolean algebras and combinatorial circuits, automata grammars and languages.)

Index

U

V

W